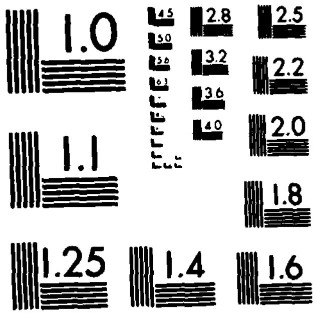
AD-A136 392 AIRBORNE INTELLIGENT DISPLAY (AID) PHASE I SOFTWARE DESCRIPTION(U) MASSACHUSETTS INST OF TECH LEXINGTON LINCOLN LAB A C DRUMM ET AL. 24 OCT 83 ATC-123 1/2

UNCLASSIFIED DOT/FAA/PM-83/30 F19628-80-C-0002 F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

(12)

Report No. DOT/FAA/PM-83/30

Project Report
ATC-123

AD-A136392

Airborne Intelligent Display (AID) Phase I Software Description

A.C. Drumm

W.S. Heath

J.A. Richardson

24 October 1983

Lincoln Laboratory

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LEXINGTON, MASSACHUSETTS

Prepared for the Federal Aviation Administration
under Electronic Systems Division Contract F19628-80-C-0002.

83 12 29 007

TECHNICAL REPORT STANDARD TITLE PAGE

1. Report No. DOT/FAA/PM-83/30	2. Government Accession No. AD-A136392	3. Recipient's Catalog No.

4. Title and Subtitle Airborne Intelligent Display (AID) Phase I Software Description	5. Report Date 24 October 1983 6. Performing Organization Code
7. Author(s) Ann C. Drumm, Walter S. Heath and John A. Richardson	8. Performing Organization Report No. ATC-123 ✓
9. Performing Organization Name and Address Lincoln Laboratory, M.I.T. P.O. Box 73 Lexington, MA 02173-0073	10. Work Unit No. 11. Contract or Grant No. DOT-FA77-WAI-817
12. Sponsoring Agency Name and Address Department of Transportation Federal Aviation Administration Systems Research and Development Service Washington, D.C. 20591	13. Type of Report and Period Covered Project Report 14. Sponsoring Agency Code

15. Supplementary Notes

This work reported in this document was performed at Lincoln Laboratory, a center for research operated by Massachusetts Institute of Technology, under Air Force Contract F19628-80-C-0002.

16. Abstract

The Airborne Intelligent Display is a microprocessor-based display capable of serving as a cockpit data terminal in a variety of FAA developmental applications. A prototype of this display was developed by Lincoln Laboratory during 1979-1980 in order to evaluate and demonstrate the use of the data link between a Mode S ground sensor and Mode S transponder-equipped aircraft. The AID served as a data link interface allowing the pilot to see, respond to, and initiate communications from a ground sensor. Later, when Lincoln began testing the Traffic Alert and Collision Avoidance System (TCAS), the AID became the TCAS display device, showing position estimates for TCAS-tracked aircraft.

More recently, a redesign effort, focused principally on software, was begun to extend the AID design so that it could be more quickly adapted to a variety of FAA developmental programs. This document describes the redesigned Airborne Intelligent Display, with special emphasis on software design.

17. Key Words		18. Distribution Statement
Microprocessor	Mode S	Document is available to the public through the National Technical Information Service, Springfield, Virgina 22161.
Display system	Software	
Airborne Collision Avoidance	Operating System	

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	148	

Form DOT F 1700.7 (8-69)

CONTENTS

CONTENTS (CONT'D)

CONTENTS (CONT'D)

CONTENTS (CONT'D)

ILLUSTRATIONS

APPENDIX ILLUSTRATIONS

1.0 INTRODUCTION

The Mode S beacon system, a combined secondary surveillance radar (beacon) and ground-air-ground data link system, is capable of providing both the aircraft surveillance and communications necessary to support Air Traffic Control automation in the future. Many uses of the Mode S data link within the FAA ATC system are apparent but are, of course, untried and need to be validated. The Airborne Intelligent Display (AID) reported here was developed by Lincoln Laboratory during 1979-1980 in order to evaluate and demonstrate the use of the data link between a Mode S ground sensor and Mode S transponder-equipped aircraft. The AID served as a data link interface allowing the pilot to see, respond to, and initiate communications with a ground sensor. Later, when Lincoln began testing the Traffic Alert and Collision Avoidance System (TCAS), the AID became the TCAS display device, showing position estimates for TCAS-tracked aircraft.

The AID is a microprocessor-based avionics display system which includes a CRT (modified Bendix color weather radar display), pilot entry device (keyboard), and annunciator.

The original AID design used a single Z80 microprocessor, assembly language coding, and ROM storage and could not be easily modified to meet growing user demands. A redesign effort, focused principally on software, was begun to develop an AID that would be flexible in responding to the needs of a variety of FAA development programs. The redesign effort is being done in phases. The phase I AID system, completed in 1982, supports the TCAS program. The phase II system will add Mode S data link capability.

This document describes the redesigned phase I AID system. Three sections follow: Section 2 covers system design, including design objectives and approach, display requirements, and hardware structure; Section 3 gives a software overview followed by general descriptions of each of the major phase I software functional units; Section 4 gives detailed descriptions of these software functional units.

2.0 SYSTEM DESIGN

2.1 Design Objectives

The objectives of the AID software redesign effort were to:

1. produce a system that could be easily adapted to future design changes and easily maintained,

2. require minimum changes to existing hardware,

3. develop software on a software development facility (SDF) that could be inexpensively duplicated elsewhere to allow the FAA Technical Center and others to develop or modify AID software,

4. use a program load device,

5. use structured, top-down software design techniques.

2.2 Design Approach

The stated objectives were met by:

1. distributing the processing load among multiple Z80 single-board computers (SBC's). This results in processing bandwidth (instructions/second) and the amount of directly addressable memory being multiplied by approximately the number of processors used. It also allows the system to be divided into logical units that can run in parallel. The software for these units may then be maintained by different organizations if the defined interface requirements are strictly observed.

2. limiting hardware changes to the addition of a second video RAM board and modification of the video controller. These changes allow more time for software screen generation and eliminate screen noise.

3. developing all software on an LSI-11 SDF and downloading object files to Z80 SDF's for testing and integration with hardware. This technique has been demonstrated to be far superior to developing software on Z80 SDF's directly.

4. using a floppy disk to permit program load before or during flight. Load time is typically 2 minutes using a single eight-inch disk.

5. writing all software in the C compiler language. This language enforces structured programming. It has been used on other similar projects and is compatible with the objectives of this project.

2.3 Display Requirements

A major objective of the AID software redesign effort was to produce an airborne display system flexible enough to respond to the needs of a variety of FAA development programs. Each development program has specific requirements in terms of equipment to be interfaced to the AID system and information to be displayed. The phase I AID system was designed to interface to a TCAS experimental unit, receiving aircraft position information and displaying targets in a Planned Position Indicator (PPI) mode on the CRT. The TCAS/AID installation was used in subject pilot tests at Lincoln. These tests gathered information on pilots' reactions to the display of TCAS traffic advisories under actual flight conditions.

Features of the phase I AID display, especially in the areas of target symbology and aural alerting, were reviewed by the FAA's TCAS II Operational Evaluation Working group. This was done to ensure that the flight testing done at Lincoln would be relevant to future TCAS installations. The phase I AID display requirements are listed below. A sample display is shown in Fig. 2.3-1.

1. Own aircraft

(a) The symbol for own aircraft will be a chevron centered horizontally on the display, approximately 2/3 down from the top of the display.

(b) Own aircraft altitude will appear in the lower left corner of the screen when the display is in absolute altitude mode. Own altitude will not appear in relative altitude mode.

2. Target aircraft

(a) Target position will be indicated by a triangle located at the range and bearing determined by the TCAS processor.

(b) An altitude tag will accompany each target, showing relative or absolute altitude (as selected) in 100's of feet. Non-mode C aircraft will display three question marks (???). An up or down arrow will indicate altitude trend whenever altitude rate is $> \pm 10$ feet/sec. The altitude tag will be in one of four positions relative to the target triangle: above, right, below, or left. Nominal position is above, but the position will be altered as required to avoid clutter with other target information.

(c) Target color will be red to indicate threat (aircraft generating a TCAS resolution advisory), amber to indicate pre-threat (aircraft generating a TCAS traffic advisory), or white to indicate proximate traffic (aircraft within 4 nm and ± 1200 feet vertically).

(d) Information for threats or pre-threats without bearing will be written in alphanumeric form in a block in the upper left portion of the screen. No indication will be provided for proximate aircraft without bearing.

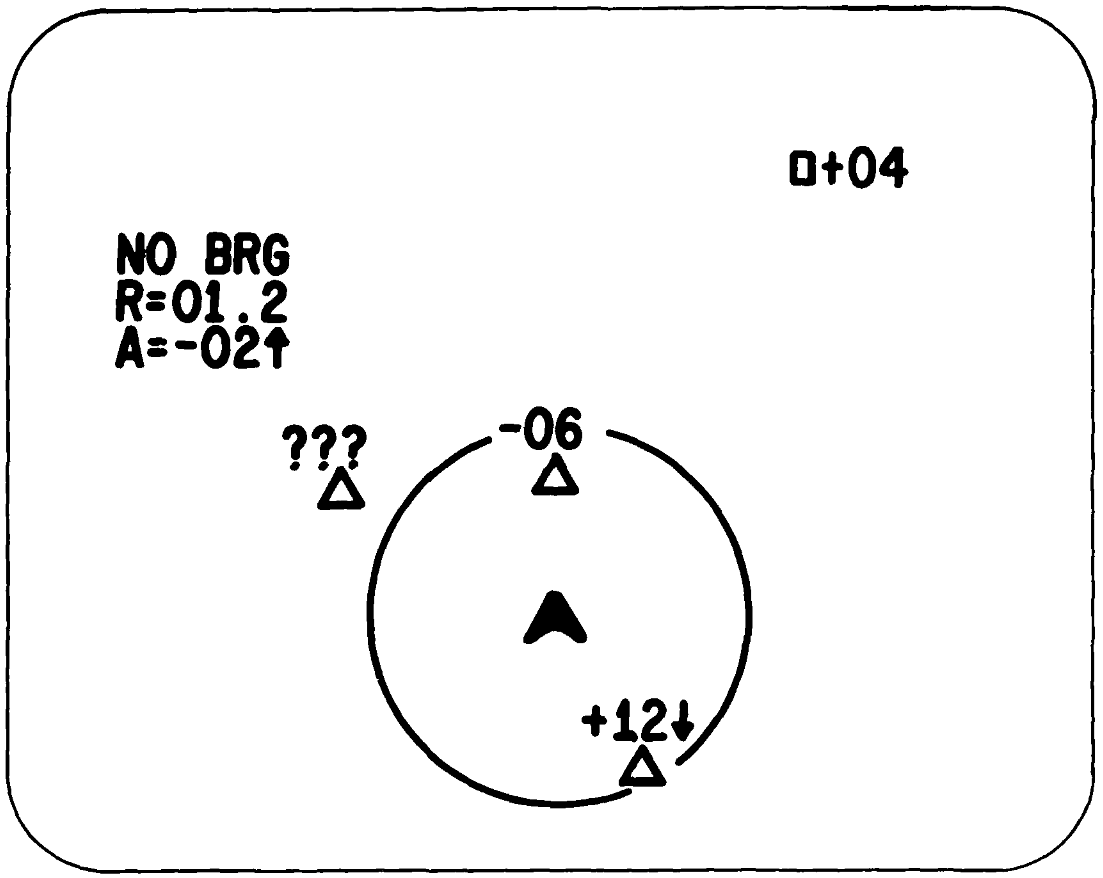


Fig. 2.3-1. Phase I Aid Display.

(e) A threat or pre-threat occurring off-screen will be indicated by a small square located at the edge of the screen at the proper bearing. An altitude tag will accompany this symbol.

[illegible]. General

(a) A software-controlled aural alerting system will be provided as specified in Appendix C. The AID will drive a visual alert light mounted on the forward instrument panel. The light will illuminate red for the appearance of resolution advisories and amber for the appearance of pre-threat traffic advisories. Pressing the light will extinguish any illuminated lights and will return a signal to the AID that the light has been pushed. Logic for control of this light is provided in Appendix C.

(b) Under normal conditions the display will be capable of providing 1 second updates for up to 8 targets with full altitude tags. But as a fail-safe feature, the display will revert temporarily to a 2 second update rate if it is ever incapable of updating all targets within 1 second.

(c) Fixed information (ownship symbol and range ring) will be overwritten (partially erased) by aircraft symbols and their associated altitude tags.

(d) A mode selector switch will be mounted in the cockpit. This switch has four positions which are described below.

Switch Position	Result
TCAS OFF	Power to AID is off. Weather radar data is displayed.
TCAS STANDBY	Power to AID is on. Weather radar only is displayed.
WX RADAR/TCAS	Power to AID is on. Weather radar data is displayed unless TCAS interrupts. Then TCAS data only is displayed for duration of interrupt.
TCAS	Power to AID is on. TCAS data only is displayed.

When the mode selector switch is in the WX RADAR/TCAS position, TCAS will interrupt whenever 1) A pre-threat or threat advisory has been generated or 2) extended display criteria are in effect.

4. Keyboard-Selectable Options

The following display options will be available for the phase 1 AID display. Default values are underlined.

(a) Range — Minimum (rear) distance.

2nm
3nm
4nm
5nm
6nm
7nm
8nm

(b) Autoscaling on/off. When the autoscaling option has been selected, the display scale will be adjusted when necessary to allow all threats and pre-threats to be visible on the display screen. One of seven scales will be selected with minimum screen distance equal to 2,3,4,5,6,7, or 8nm. Regardless of the autoscaling option selected, the selected fixed display scale will be used whenever this scale allows all threats and pre-threats to be visible on the display screen.

(c) Altitude format — relative altitude / absolute altitude

(d) Proximity suppression — suppress proximity advisories (triggered mode) / display proximity advisories (continuous mode)

In triggered mode, proximity advisories are suppressed except when threat (red) or pre-threat (amber) advisories are present. The display resuppresses 8 seconds after all threats and pre-threats have cleared. In continuous mode, advisories (including proximity advisories) are displayed whenever tracks qualify.

(e) Display criteria — normal criteria / expanded criteria (call-up mode)

A "call-up" button will be provided on the keyboard which can temporarily expand the display criteria. If the display is in triggered mode, then pressing the call-up button results in unsuppressed display for 15-seconds. During this time all proximity advisories will be displayed. If the display is in continuous mode, then pressing the button results in display of all tracks within 4 miles and 1200 feet for a 15 second period. No off-screen symbols will be generated for targets which satisfy only expanded display criteria.

(f) Number of targets to display. 0 - 8

The TCAS logic will provide priority ranking for all targets sent to the AID. This ranking will be used to delete targets when the display limit is exceeded.

2.4 Hardware Structure

2.4.1 Overview

Figure 2.4-1 is a block diagram of the AID hardware configuration. Phase I components are shown enclosed in solid lines. Components to be added for phase II are shown enclosed in dashed lines.

The system is partitioned into functional units by the use of multiple single-board computers (SBC's). The SBC's are connected in a master/slave configuration. The master SBC, referred to as the service processor, serves primarily as a general-purpose audio/video processor. One or more slave SBC's serve as user processors, performing functions which are specific to a particular user application. All slaves communciate with the master via the S-100 buss. The master then interfaces with the CRT, audio annunciator, and floppy disk. Unlike the slaves, the master has complete access to the S-100 buss. The slaves use the S-100 data lines and some status and control lines in their communication with the master. However, they cannot put an address on the S-100 address lines. In this sense they are similar to I/O controller devices on the buss.

The sections which follow describe briefly the hardware components shown attached to the SBC's in Fig. 2.4-1. The hardware discussion ends with sections on S-100 Slot Usage and SBC Characteristics.

2.4.2 Video RAM, Video RAM Controller, and Video Multiplexer

The video RAM controller has been redesigned to add a second video RAM. In the original AID, the video controller and the CPU accessed a single video RAM card. This card contained three 8K banks of RAM -- one each for red, green, and blue data. Normally the screen was blanked while the computer updated the video RAM. This caused a noticeable blink on the screen. In addition, when symbols were purposely blinked, both the computer and the video controller accessed common data lines, causing noise on the screen. To correct these problems it was decided to use two video RAM cards and to add a video multiplexer. In this way the video controller can be reading a screen image from one video RAM while the CPU is loading the other. The controller can then switch to the updated image in the other RAM during the vertical retrace of the CRT. This eliminates all blinking and noise problems. It also provides the CPU with a full video frame period to generate a new frame. The additional video RAM card is identical to the original unit.

2.4.3 Audio RAM and Audio Annunciator

The phase I AID system uses three 16K banks of audio RAM for storing words and phrases to be annunciated. The three banks used are the upper 16K of the master's onboard memory plus two 16K banks from a 64K RAM card. The 16K off-board RAM banks are selected by the master by de-selecting the onboard bank with the same address space. The master loads selected words or phrases from these audio banks into an annunciator RAM, then activates the annunciator.

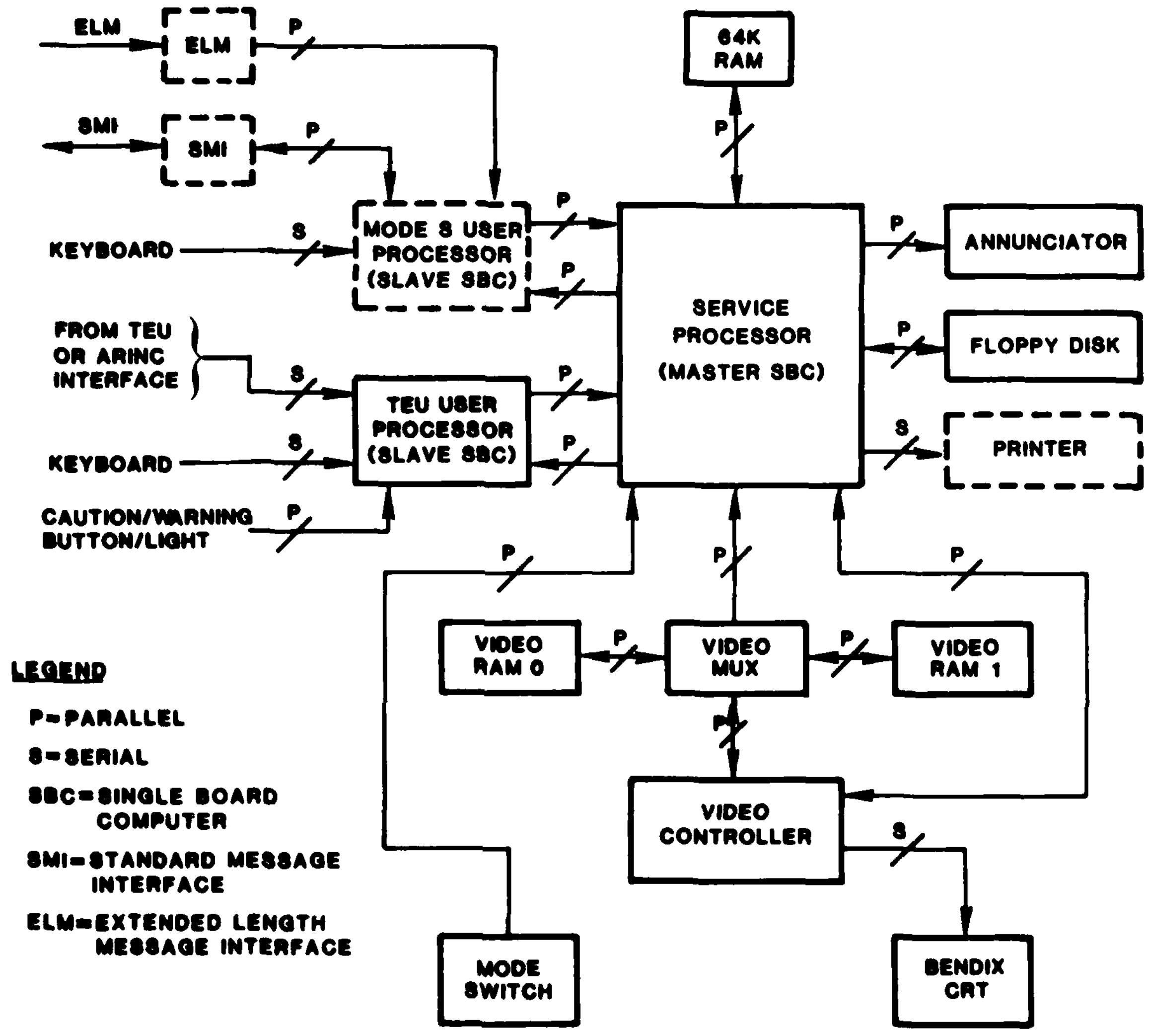

ELM
ELM
P
SMI
SMI
P
64K RAM
P
KEYBOARD
S
MODE S USER PROCESSOR (SLAVE SBC)
P
P
SERVICE PROCESSOR (MASTER SBC)
P
ANNUNCIATOR
P
FLOPPY DISK
S
PRINTER
FROM TEU OR ARINC INTERFACE
S
TEU USER PROCESSOR (SLAVE SBC)
P
P
KEYBOARD
S
CAUTION/WARNING BUTTON/LIGHT
P
P
P
P
VIDEO RAM 0
P
VIDEO MUX
P
VIDEO RAM 1
P
VIDEO CONTROLLER
S
MODE SWITCH
BENDIX CRT
LEGEND
P=PARALLEL
S=SERIAL
SBC=SINGLE BOARD COMPUTER
SMI=STANDARD MESSAGE INTERFACE
ELM=EXTENDED LENGTH MESSAGE INTERFACE

Fig. 2.4-1. AID hardware configuration.

To be consistent with the design philosophy, the audio data should be kept in the appropriate slave, since it is application dependent. However, this would be inefficient since audio data would then have to be sent back to the master with each audio command. The audio data file can still be maintained by the same person or group supporting the application. If multiple applications exist in separate slaves, then multiple audio data files can exist and can be loaded into the same or different RAM banks.

2.4.4 Floppy Disk

Each SBC has 64K of onboard RAM. ROM is used only for boot program storage; program and data files are stored on floppy disk and loaded into the onboard RAM. The phase I AID system contains a single floppy disk drive and uses 8-inch, single-sided, double-density floppy disks for storage. Because only the master interfaces to the floppy disk, the master must be responsible for the loading and proper distribution of all program and data files. In response to a system boot, the CP/M operating system is loaded into the master. CP/M then in turn automatically loads the program and data files into the master. From there, slave programs and audio data files are downloaded via the S-100 buss to the proper destinations.

Disk access is required only during initial program load. Following this, the floppy disk can be removed from the drive and those parts of CP/M that handle disk access can be overwritten by the master's application program.

2.4.5 Mode Switch

The Bendix front panel mode switch is used to switch the video display among four positions: test, weather radar only, combination weather radar/AID, and AID only. The switch is interfaced to one of the master's parallel ports on top of the card. When there is a change in the switch position, the switch interrupt handler causes the new switch position to be sent to all slaves. The slaves can then change their operation as necessary.

2.4.6 Caution/Warning Button/Light

A combination button/light is interfaced to the phase I TCAS slave via one of the two parallel ports on top of the card. The upper half of the button contains a red light labeled 'warning'; the lower half contains an amber light labeled 'caution'. Software in the slave turns on one of the lights and annunciates a corresponding audio phrase when warranted by the aircraft threat environment. When the user presses the button, an interrupt is generated in the slave. The interrupt handler then extinguishes both the light and the audio annunciation.

2.4.7 Keyboard and TEU Serial Input

Serial inputs to the phase I TCAS slave are from the aircraft's onboard TCAS Experimental Unit (TEU) and the keyboard. These use the two serial ports on the top of the card.

2.4.8 S-100 Slot Utilization

Figure 2.4-2 diagrams buss slot usage for the AID design. Again phase I components are enclosed in solid lines; phase II components are enclosed in dashed lines. The figure also shows card interconnections via connectors on the tops of the cards.

Note that an additional slave SBC and an ARINC 429 interface card are shown. Their purpose is to convert ARINC-formatted TCAS data into a format compatible with the current TEU-AID RS-232 interface. This provides an interface to Dalmo Victor's TEU equipment. Note that these two cards will only draw power from the S-100 buss. All communications are through connectors on the tops of the cards. They therefore have no effect on the AID cards on the buss. This SBC's programs are burned into EPROM's.

The ARINC interface also serves a second purpose. If the ARINC TEST OUTPUT is connected to the ARINC INPUT, then ARINC test messages, generated within the ARINC slave, can be sent to the TCAS slave SBC. In this way all features of the audio/video display can be tested/demonstrated.

2.4.9 Single-Board Computer Characteristics

The salient characteristics of the master and slave SBC's are summarized below. The SBC's are supplied by Sierra Data Sciences, Fairview Park, Ohio.

The master SBC:

1. uses the Z80A (4-MHz) processor,
2. contains 64K bytes of RAM divided into four 16K banks,
3. contains 4K of "shadow" EPROM (that is, the EPROM shares the 64K RAM address space. It can be switched in or out. It contains a boot program to load CPM from the disk),
4. is compatible with IEEE-696 buss standard (i.e., the IEEE standard for the S-100 buss),
5. has two serial and two parallel I/O channels accessible from the top of the card and,
6. has four counter-timers.

The master communicates with the slave SBC using the same protocol that it would use to communicate with any other I/O controller (slave) device. This protocol has been standardized by IEEE-696.

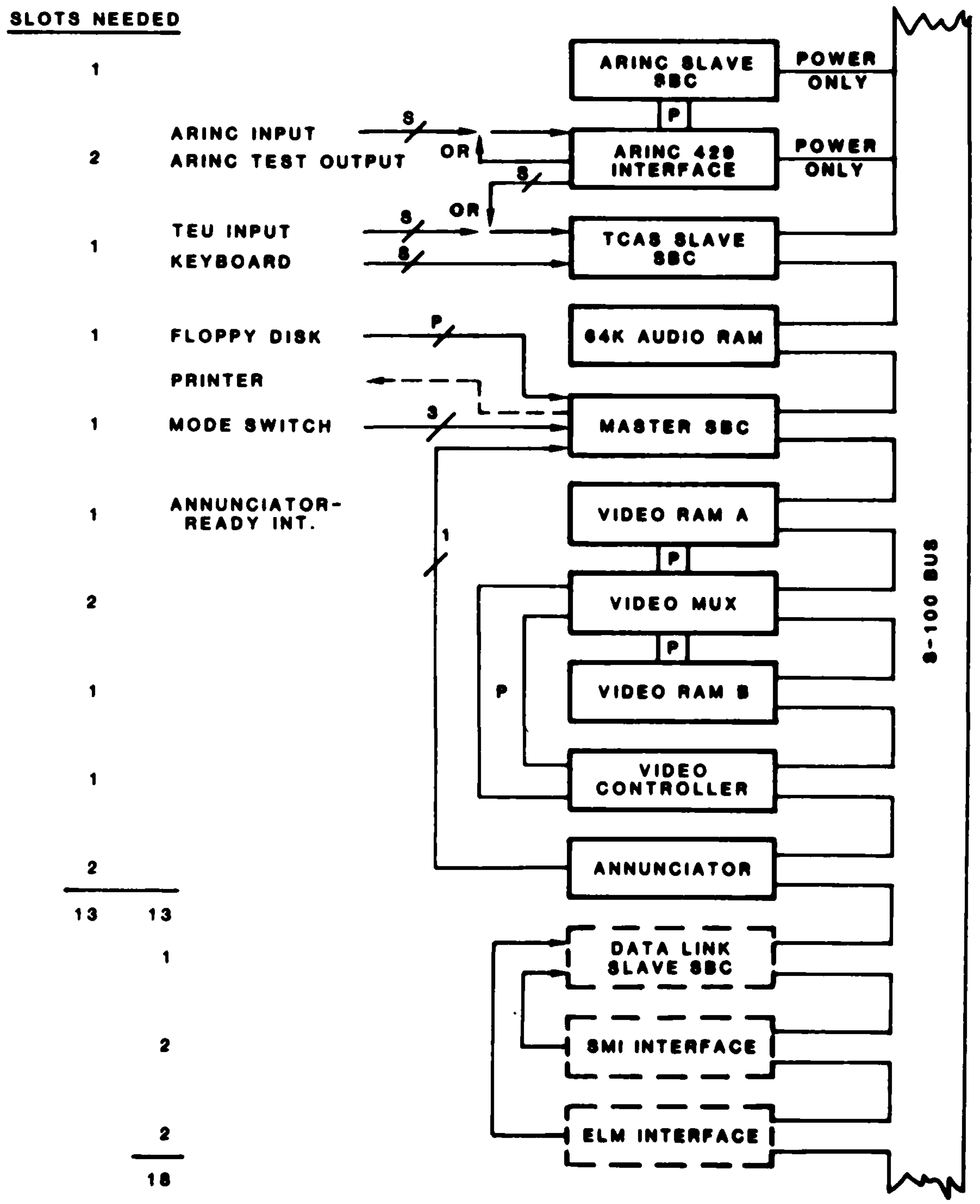


Fig. 2.4-2. S-100 buss slot usage.

The slave SBC:

1. uses the Z80A (4-MHz) processor,

2. contains 64K bytes of RAM divided into four 16K banks,

3. contains up to 16K of "shadow" EPROM,

4. has two serial and two (or four, optional) parallel channels,

5. has four counter-timers and

6. contains an X-buss expansion interface.

The X-buss contains lines from all Z80A pins plus additional control and status signals generated on the board. It may be used to interface to another memory bank or to a utility card containing a high-speed math chip and additional serial and parallel ports.

The slave is supplied with a single 2732(4K) EPROM containing a boot program which causes the CPU to wait for a program download from the master.

3.0 SOFTWARE GENERAL DESCRIPTION

3.1 Overview

The AID software, like the hardware, is partitioned into functional units by the use of multiple single-board computers (SBC's). The master SBC, referred to as the service processor, serves primarily as a general-purpose audio/video processor. One or more slave SBC's serve as user processors, performing functions which are specific to a particular user application. Some software, called system software, is common to all SBC's.

The phase I software described in this document provides for a single-user application and thus contains a single-user processor. This user processor interfaces to a TCAS experimental unit (TEU) and a keyboard. Its function is to input TEU aircraft position information, process the information according to keyboard commands, and generate and send audio and video data blocks to the service processor. The service processor then drives the audio annunciator and CRT to produce audio and video output. Phase I audio outputs are of two types: (1) tones to indicate whether valid or invalid keys have been pressed on the keyboard, and (2) words or sounds to inform the pilot of a recommended maneuver or simply draw his attention to the display. Video output is a color PWI-type display showing targets at given ranges and bearings from own aircraft which is located near the center of the screen.

As stated earlier all programs are written in C. The C compiler allows direct machine code (object bytes) to be inserted in-line in a C program. The direct code can reference C-defined parameters. This machine code is used in some cases to program I/O interfaces when timing constraints require very efficient coding.

All programs are composed of tasks and interrupt handlers. A nonpre-emptive task scheduler satisfies the requirements of this program. Communication between tasks and between interrupt handlers and tasks is by means of circular queues. All programs use the same task scheduler design and queue management functions (i.e., functions for entering messages into queues and removing messages from queues).

Certain naming conventions have been followed. In general, interrupt handlers contain letters of the attached device followed by IN or OUT depending on the direction of the data flow. Tasks which serve a function similar to that of the corresponding interrupt handler are distinguished from the interrupt handler by an additional letter T. Queue names generally contain 6 letters, the first three corresponding to the function which inputs data to the queue, the last three to the function which removes data from the queue.

The three subsections which follow give a general description of the phase I software: Section 3.2 - System Software, Section 3.3 - Service Processor Software, and Section 3.4 - User Processor Software. A more detailed description of the software in each of these areas is given in Section 4, Software Detailed Description.

3.2 System Software

Topics to be discussed in this section include interprocessor coordination functions: system startup (3.2.1) and interprocessor communication (3.2.2); functions used in common by all application programs: the task scheduler (3.2.3) and queue management functions (3.2.4); and diagnostics (3.2.5).

3.2.1 System Startup

When an SBC is initially booted (power turned on) it runs a boot program stored in an on-board EPROM. The slave's boot program initializes the slave to receive a program download from the master. The master's boot program loads the CPM operating system from tracks zero and one of the floppy disk.

This section discusses the initial program load procedure, application program initialization, and interprocessor startup coordination.

3.2.1.1 Initial Program Load

Except for boot program storage, all memory in the AID is RAM. The CPM operating system plus program and data files are stored on and loaded from floppy disk.

The Sierra Data Sciences' system configuration utility has been used to modify parts of the CP/M operating system residing on the floppy disk. Specifically, an autoload command line has been specified. This command line contains a list of simulated operator commands. When CP/M is initially loaded into the master, it checks to see if this command line is present, and if so, executes the first command. Upon completion, the system does a warm boot and executes the next command in the command line. This procedure continues through execution of the last command in the command line.

The first three commands load the three 16K banks of audio data into the master's upper 16K memory bank and into banks A and B of the 64K RAM card, respectively. The fourth command loads a download program into the TCAS slave. The fifth command loads a corresponding download program into the master. Together the master and slave download programs then read the slave application program, one block at a time, from the floppy disk into the master, then send the program, still one block at a time, to the slave. Once loaded, the slave application program begins execution. The sixth and final command loads and executes the master's program. Note that once the master's application program is loaded and begins running, control is never returned to CP/M. The master program may therefore overwrite CP/M in the Master's RAM memory.

During system integration and at other times for troubleshooting, it is desirable to run debuggers in both master and slave. Terminals are attached to the master and slave serial ports to support this, and software is changed in the slave application program to configure one of the serial ports for the terminal. (In normal operation, the two slave serial ports are used for TEU and pilot keyboard inputs. For debugging the slave serial ports are used for either terminal and TEU or terminal and pilot keyboard.) The download of the slave application program is accomplished in the same manner as described above except that a different download program is loaded into the slave. After this download program finishes downloading the application program, it transfers control to Sierra Data Sciences' slave monitor program instead of to the start of the application program. In the master, instead of directly loading the master program, the standard CP/M symbolic debugger ZSID is loaded. The master program is then loaded under ZSID control.

3.2.1.2 Program Initialization

Each program loaded into an SBC goes through a similar initialization sequence. All C programs begin with the function main(). In this application the initialization process is divided into two parts: that performed by the function main() and that performed by the task init().

The main() function performs all initialization operations necessary only at startup and not during a restart. These include:

a) zero the data area
b) load the interrupt vector table
c) initialize the task scheduler
d) initialize all task control blocks (TCB)
e) initialize all tasks (call them and run them to their first suspend point)
f) schedule the init() task
g) call the task scheduler.

The init() task performs all functions necessary to perform a restart. These include:

a) initialize all circular queue headers
b) initialize all hardware I/O devices (e.g., the counter-timers, parallel and serial I/O chips, audio and video controller boards, etc.).

Note that the main()/init() partitioning of initialization is more appropriate in a system in which the program is stored in ROM. In that case the application programs can initiate a restart by scheduling the init() task. Since the program is in ROM it is likely that this process will be successful. However, if the program is in RAM, it is possible that the program itself was altered during abnormal operation, and restart will not be successful. Initialization is partitioned as described so that the program will be suitable for ROM storage if the need should arise at a later date. In the phase I system, restart is accomplished by rebooting the entire system from the disk.

3.2.1.3 Interprocessor Startup Coordination

Slaves complete initialization before the master and wait for a "POKE" message from the master. While waiting, slave external interrupt handlers ignore all data received (i.e., will not wake tasks to process data). When a

POKE is received, the slave activates its external interrupt handlers and sends an acknowledgement back to the master. The master sends POKE messages to each slave. When all slaves have replied, the master activates the front panel mode switch handler and sends the current switch setting to all slaves. The switch setting indicates which slave should send audio/video data to the master. This completes the startup process.

3.2.2 Interprocessor Communication

Messages sent between master and slave have fixed formats. The first two bytes of each message contain a type code and a byte count, respectively. An actual transmission can be a string of concatenated, fixed-format messages. Each transmission of a set of messages is initiated by sending a single byte containing the total number of bytes to follow (up to 255). The block of concatenated messages of the indicated length is then sent.

The message formats are shown in Figs. 3.2-1 through 3.2-4. There are four general message categories: video control, video graphics, audio, and miscellaneous.

Figure 3.2-1 shows video control messages. All messages describing a single video frame must be preceded by a START-OF-FRAME message and followed by an END-OF-FRAME message. The screen is blanked by sending a CLEAR message. An initial SCALE message is sent from the user processor to the service processor to specify the dimensions of the video screen. (See Section 3.4.4 for a description of the virtual screen concept.)

The remaining three video control messages (COLOR, LINE TYPE, and REVERSE VIDEO) each select an option which then remains in effect until changed by a CLEAR command or the video control message with a different option selected. The control byte in the COLOR message selects one of seven colors. The control byte in the LINE TYPE message selects dashed, dotted, or solid lines. The control byte in the REVERSE VIDEO message selects either reverse video on or off (off = normal mode).

Three video graphics message types have been defined: STRING, CIRCLE, and LINE. These are shown in Fig. 3.2-2. The STRING message specifies the X,Y starting coordinates of an ASCII character string, a reference position for the first character (i.e., centered on X,Y, lower left at X,Y, etc.), and the ASCII string itself. The ASCII string is limited to 32 characters, the width of the screen for our application. The CIRCLE message gives the X,Y coordinates of the circle center and its radius. The LINE message contains a byte specifying the number of line segments to be drawn and the X,Y coordinates of the lines.

Figure 3.2-3 shows the three types of audio messages. The basic AUDIO message specifies an offset into the audio RAM data area and the length of that data area in bytes. Multiple AUDIO messages may be combined to form a single phrase by preceding the AUDIO messages by a START-OF-AUDIO and

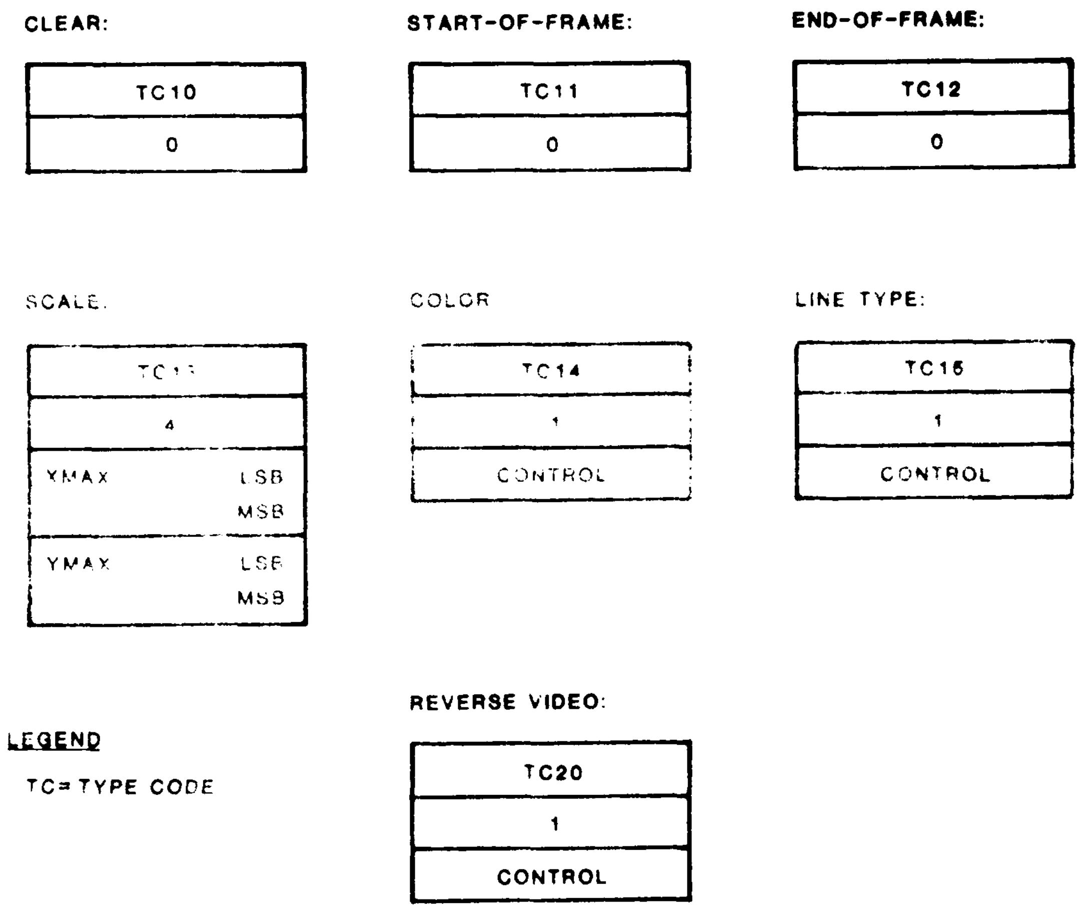


Fig. 3.2-1. Video control messages.

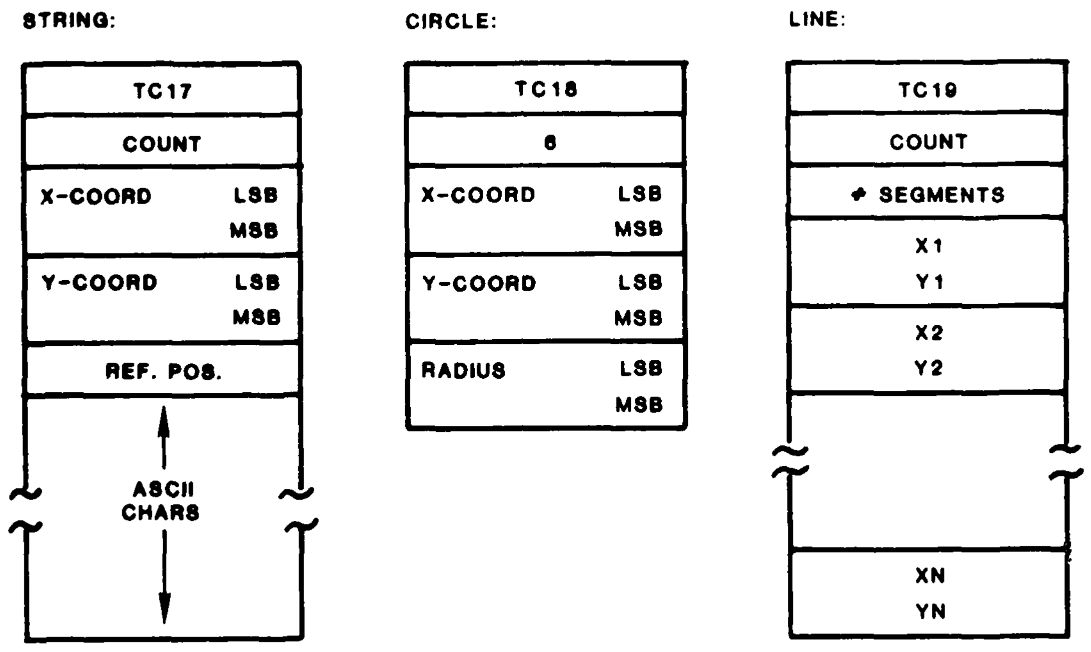


Fig. 3.2-2. Video graphics message.

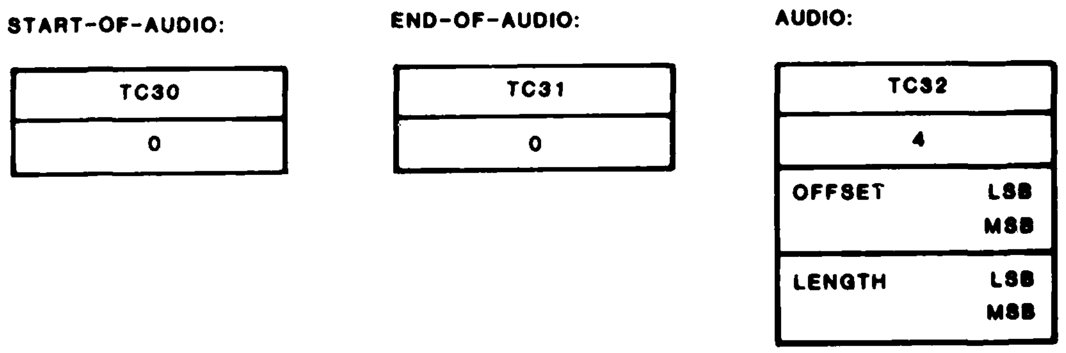


Fig. 3.2-3. Audio messages.

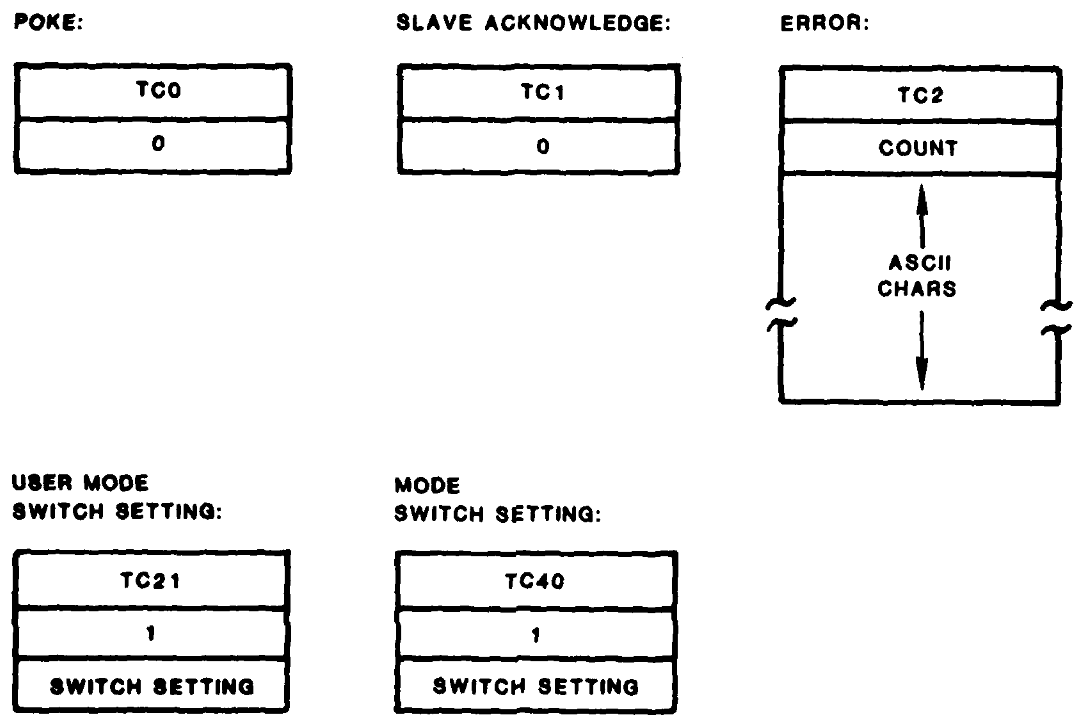


Fig. 3.2-4. Miscellaneous messages.

[illegible] them [illegible]. All such enclosed ACD10 messages are [illegible] message is [illegible] message [illegible].

[illegible]

[illegible] priority [illegible] data and is used [illegible] master [illegible] switch is in the [illegible]. The master [illegible] determines whether [illegible] display [illegible] data [illegible] the [illegible] MODE [illegible].

3.2.3 Task Scheduler

The application software is broken into functional blocks called tasks. Each task carries out a specific function. Each task is written as a sequential program, i.e., proceeding from beginning to end without a break in execution. At any one time, several tasks may be ready to run. Since only one task can be executing at a given time, the scheduler performs the function of determining which of the ready tasks to execute.

Figure 3.2-5 shows the three possible states of a task: Ready, Running or Waiting. A Waiting task is stopped because it is waiting for some condition to occur, such as receipt of an input character. A Ready task is ready to run but is stopped because another task is executing. The Running task is the currently executing task. The transitions between the various states are triggered by the run(), sleep() and wake() functions. The scheduler executes the run() function to start a task running. The Running task will execute the sleep() function when it reaches a point in the execution where it must wait for some condition, such as keyboard input. Sleep() saves the task's stack pointer in its Task Control Block (TCB) and [illegible] control [illegible] to the task scheduler. The scheduler then determines the [illegible] task to be executed from the list of Ready tasks. Tasks are transferred from Waiting to Ready by the wake() function. Wake() can be issued by an interrupt handler or by another task.

Each task has an associated Task Control Block as shown in Fig. 3.2-6. [illegible] entries include the task Status flag, Signal flag, stack pointer, and pointer to the next TCB. The Status flag, when set, indicates that the task [illegible]. The Signal flag indicates that the task has been awakened by an interrupt handler or another task. The purpose of the Signal flag is to

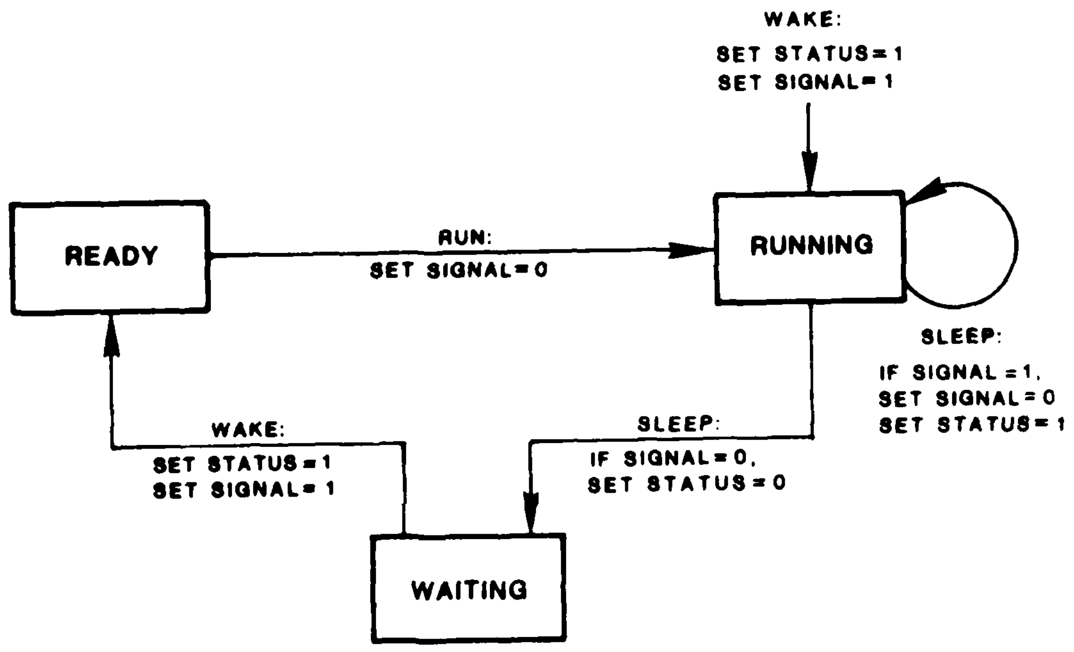


Fig. 3.2-5. Task states.

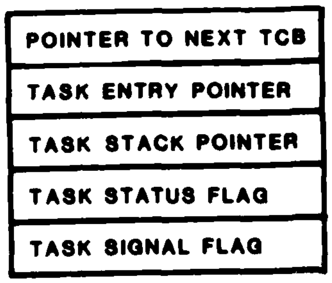


Fig. 3.2-6. Task control block.

prevent a task wakeup from being lost while the task is running. The stack pointer is used to save return addresses for tasks that are Waiting or Ready. Each task has its own stack. The pointer to the next TCB points to the next highest priority task.

When a task is awakened, the Status and Signal flags are both set. The task is now in the Ready state. When a currently running task executes the sleep() function and enters the Waiting state, the scheduler examines the TCBs to determine the highest priority task that has the Status flag set. The scheduler then clears the Signal flag of this task and starts the task running. If the task is awakened while running, the Signal flag is set, so that wake ups are not lost while a task is executing. The sleep() function always checks the signal flag of the Running task before suspending execution of the task. If set, sleep() clears the signal flag and resumes executing the Running task.

The application software functions operate below the executive level. They are implemented as re-entrant tasks. When a task is running, it cannot be suspended by another task. This type of task scheduling is termed nonpre-emptive since a higher priority task cannot pre-empt a running task. Task execution is suspended when a hardware interrupt occurs but the running task is restored when the interrupt service is complete. This type of task scheduling avoids complex problems associated with inter-task data transfer. However, it also means that higher priority tasks can be locked out by lower priority tasks. For this reason, tasks must be designed to cooperate in their use of available processing time.

3.2.4 Message Queue Management

After initialization all intertask and interrupt handler/task communication is performed by means of queues. Since tasks run asynchronously this assures that messages will not be lost (over-written). Since the messages required in this application are variable-length, the queue entries are also variable length. The same queue management functions are used by all application programs.

The queue access functions are written so that when an attempt is made to enter a message in a full queue, the task is suspended. Later, when a message is removed from the queue, the suspended task is awakened so that it can store its message. Similarly, when a task attempts to remove a message from an empty queue it suspends. When a message is later placed in the queue the suspended task is awakened. In this way messages are "gated" through the program.

3.2.5 System Diagnostics

There are two forms of AID system diagnostics: non real-time and real-time. Non-realtime diagnostics are stored on their own floppy disks and run separately from the application program, either routinely to perform system checkout or specifically to pinpoint a suspected malfunction. In contrast, realtime diagnostics are part of the application program. They monitor actual system operation.

3.2.5.1 Non-Realtime Diagnostics

The reliability of the AID hardware has been excellent. There have been no known failures in any hardware components. Therefore the only diagnostics run on a regular basis are floppy disk diagnostics. Two in-house programs exist, TFLOP - test floppy disk, and WFLOP - write floppy disk. Both programs communicate with the floppy disk controller chip and print out any unusual status conditions which occur during disk operations. The programs are interactive and user-friendly, guiding the user through selection of a variety of options for testing the health of the entire floppy or a specific area only. In practice, TFLOP performs all necessary tests. WFLOP is not normally used.

A diagnostic package was purchased which runs under the CP/M operating system and is designed to test each major component of a CP/M-based Z80 microprocessor system. These components include memory, CPU, disk drives, CRT terminal, and printer. We are not currently using this diagnostic package. It requires modification to run successfully with our Sierra Data Sciences equipment, but it is available as a starting point should some of these tests be considered necessary in the future.

3.2.5.2 Realtime Diagnostics

The AID realtime diagnostics operate in one of two ways. (1) The system detects an error condition and sends a message either to the CRT or the printer. (The printer is not implemented in phase 1.) (2) The user selects a test mode of operation (e.g., presses the TST key on the keyboard), then checks to see that the audio and video outputs are correct. The method described in (2) checks the performance of the system as a whole. The error checks used in method (1) are present and operational at all times, whether the software and hardware are in special test modes or not. These checks catch more specific errors that might not be apparent from simply observing the system audio and video outputs.

The error conditions currently printed on the CRT include (1) 'user inactive' - i.e., slave not responding to POKE messages from the master, (2) 'no data' - no TEU input received for 8 seconds, and (3) 'bad input' - TEU input fields do not pass reasonableness checks. Many other error conditions are sent from the slave to the master intended for the printer. These include checksum errors in input data, queue overflows, timing conditions that should never occur. These error checks are currently present only in the slave. In phase II, when the printer is available, error checks will be included throughout the master as well.

There are currently four test states in which the system can be run and observed. In all four states, user interactions via the keyboard and the caution/warning button function normally. For test states (3) and (4) refer to the AID Hardware Block Diagram (Fig. 2.4-1) and the S-100 Bus Slot Usage (Fig. 2.4-2).

(1) By pressing the TST key on the keyboard, the user selects a mode of operation in which canned data for 8 targets is input once per second to the user processor's TEU task for processing. Each target's range, bearing, color, and associated audio are updated each second in a realistic manner. The data repeats approximately every three minutes.

(2) When the TST mode is used in combination with the DEMO key, the user can select one of eleven fixed target scenarios or one of ten moving encounters showing own aircraft with one or two intruder aircraft. (See Fig. 3.4-2 for operating details.)

(3) By changing the cable which plugs into the AID system's TEU input port, the user can input actual recorded flight data for processing. TEU inputs from four encounters were recorded onto floppy disk. A separate single-board computer runs a program which reads the data from disk, then sends it at one-second intervals over an RS-232 link to the user processor.

(4) A shorting plug can be used to route test data from the ARINC slave single-board computer into the ARINC 429 interface to be sent to the user processor TEU input port. This provides yet another set of audio and video outputs which can be observed.

3.3 Service Processor Software General Description

The service processor is intended to be a general-purpose processor in the AID system. It acts as bus master and is responsible for controlling the AID display hardware, the audio annunciator system, and other utility devices, such as the cockpit printer, thus allowing the user processor to concentrate on its particular application. Since the service processor is the bus master, it also has responsiblity for loading the user software into the user processor.

3.3.1 Overview

The primary function of the service processor is the control of the video display so that the user processor need not be concerned with the details of driving the display. A set of general-purpose graphic commands are provided by the service processor in order to allow a user to easily generate graphic and alphanumeric displays. The service processor also provides control of the audio annunicator hardware, thus relieving the user from the details of directly handling the device. Additionally, the service processor can provide support for other utility devices which may be required by user applications.

The software in the service processor consists of several interrupt handlers, a task scheduler, and several utility tasks (see Fig. 3.3-1). Functionally, the service processor receives commands from the user processor; these are then dispatched to the appropriate task for execution (see Fig. 3.3-2). In addition, the service processor sends status, commands and data to the user processor, depending upon the configuration required by the user software and hardware.

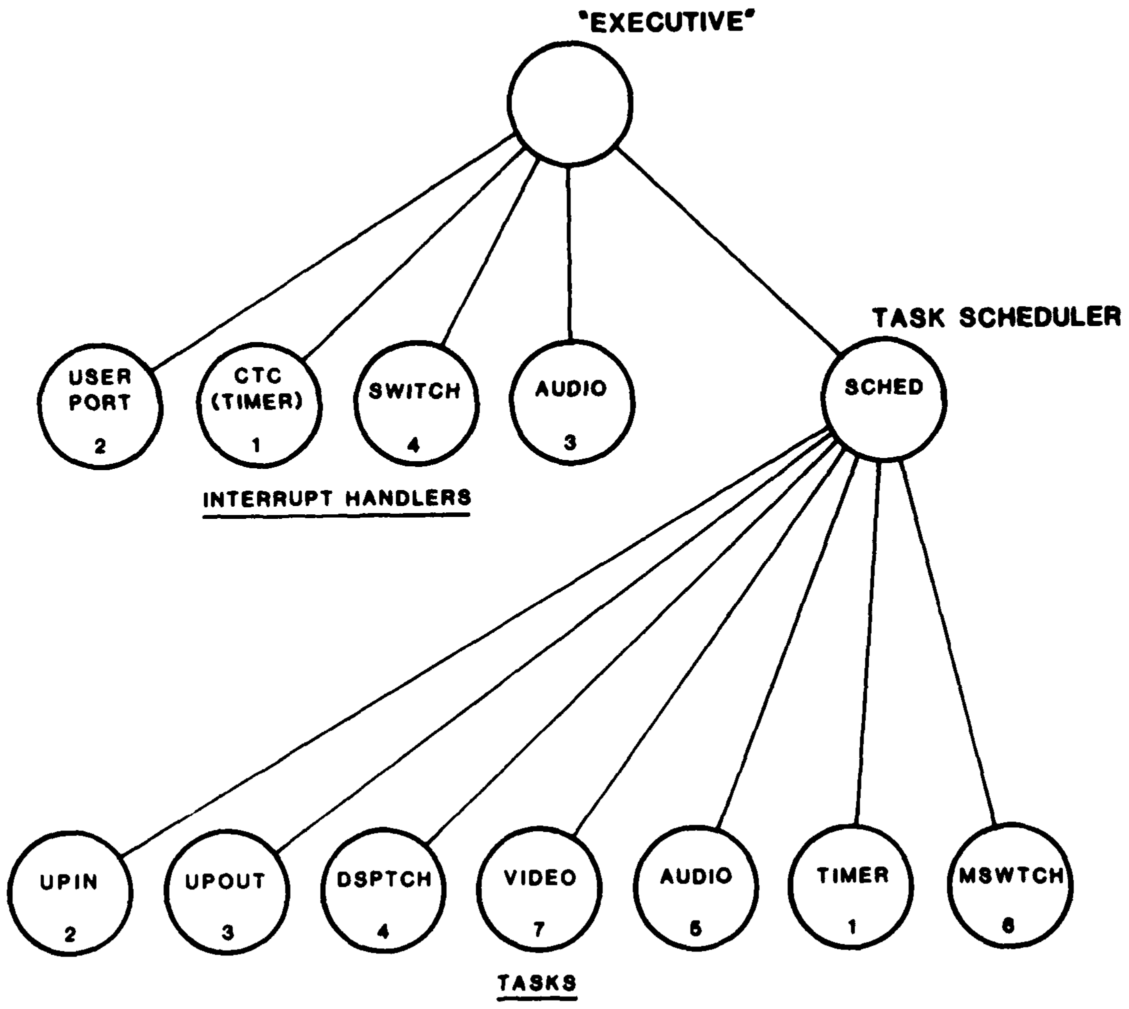


Fig. 3.3-1. Service processor interrupt handlers and tasks.

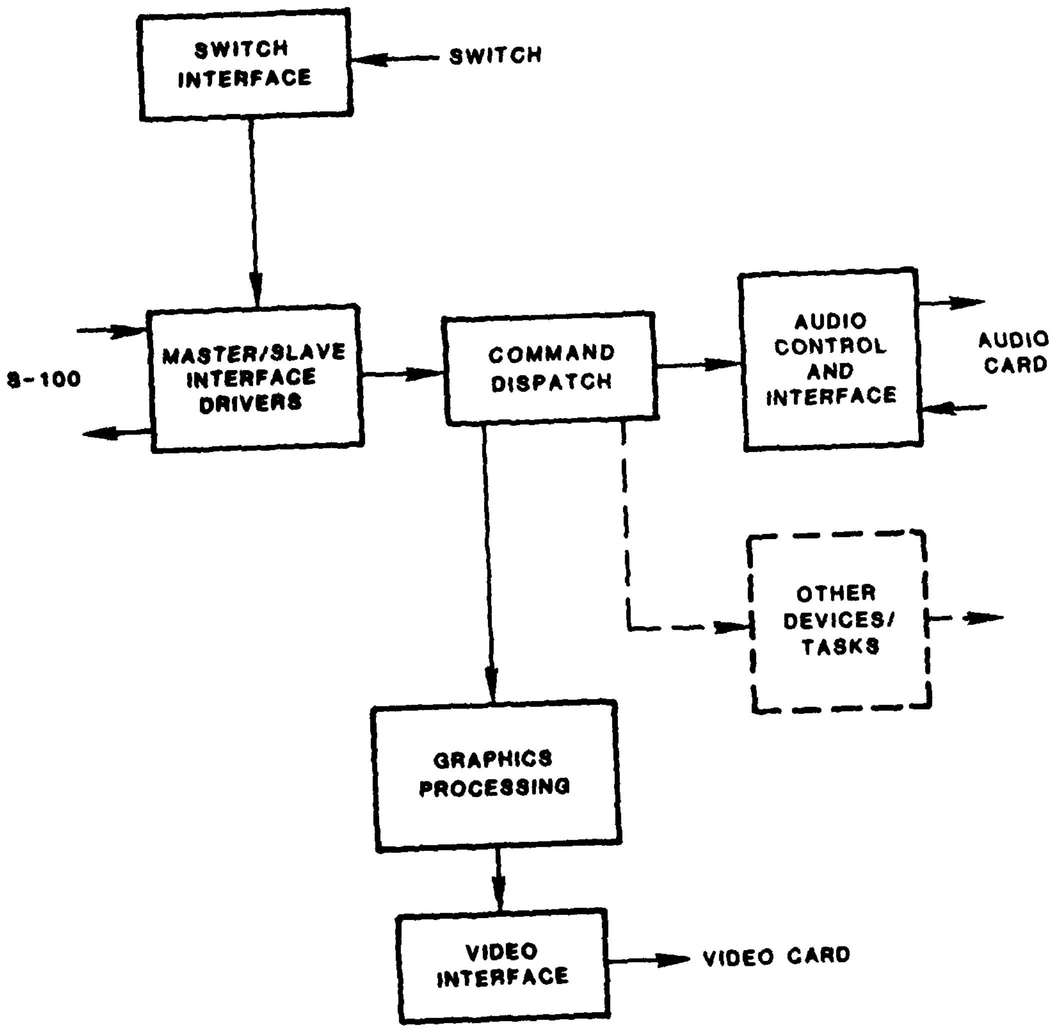


Fig. 3.3-2. Service processor functional block diagram.

Communication among tasks and between tasks and interrupt drivers is accomplished by means of first-in-first-out queues. The flow of data between interrupt handlers and tasks and between tasks is shown in Fig. 3.3-3. The task scheduler is nonpre-emptive and runs tasks on the basis of priority.

3.3.2 Interrupt Handlers

Since the service processor may be required to handle devices needing fast response, it is necessary to minimize interrupt latency. To this end, a multi-level, prioritized interrupt structure has been implemented, requiring that low-priority interrupts, which require appreciable processing, be interruptible. In general, therefore, most interrupt handlers consist of a task to perform time-consuming computations, and a very short, fast interrupt service routine.

The numbers under the interrupt handlers shown in Fig. 3.3-1 indicate the relative priority of the interrupts, with 1 being the highest.

3.3.2.1 The Service Processor/User Processor Communications Interface

Data transfers between the service processor and the user processor consist of variable-length blocks, the first byte of which contains the count of the subsequent bytes in the transfer, thus constraining transfers to less than 256 bytes. The first byte, containing the count, also acts as a *handshake signal, allowing the two processors* to synchronize the transfer. Each transfer direction is fully independent, requiring a separate interface driver and task to manage the transfers (see Fig. 3.3-2).

Messages received from the user processor are initiated by an interrupt from the user processor port. The interrupt handler upin() passes this one byte message to upint(), the user processor input task, which receives the remaining bytes of the transfer. Messages sent to the user processor are output directly by task upoutt().

3.3.2.2 The Timer Interrupt Handler (ctcin())

The counter-timer circuit interrupt handler, ctcin(), receives an interrupt whenever the timer counts down to zero. This interrupt merely puts a message into the timer() task input queue and wakes the timer() task. It is the responsibility of the timer() task to determine what actions, if any, this event triggers.

3.3.2.3 The Mode Switch Interrupt Handler (bswin())

The mode switch interrupt handler monitors parallel port lines which are connected to the Bendix front panel mode control switch. Whenever the state of this switch changes, an interrupt is generated. The state of the switch is read by bswin() and passed to the mode switch task, bswtch(). The parallel port is then reconfigured to respond to any change from the new switch setting.

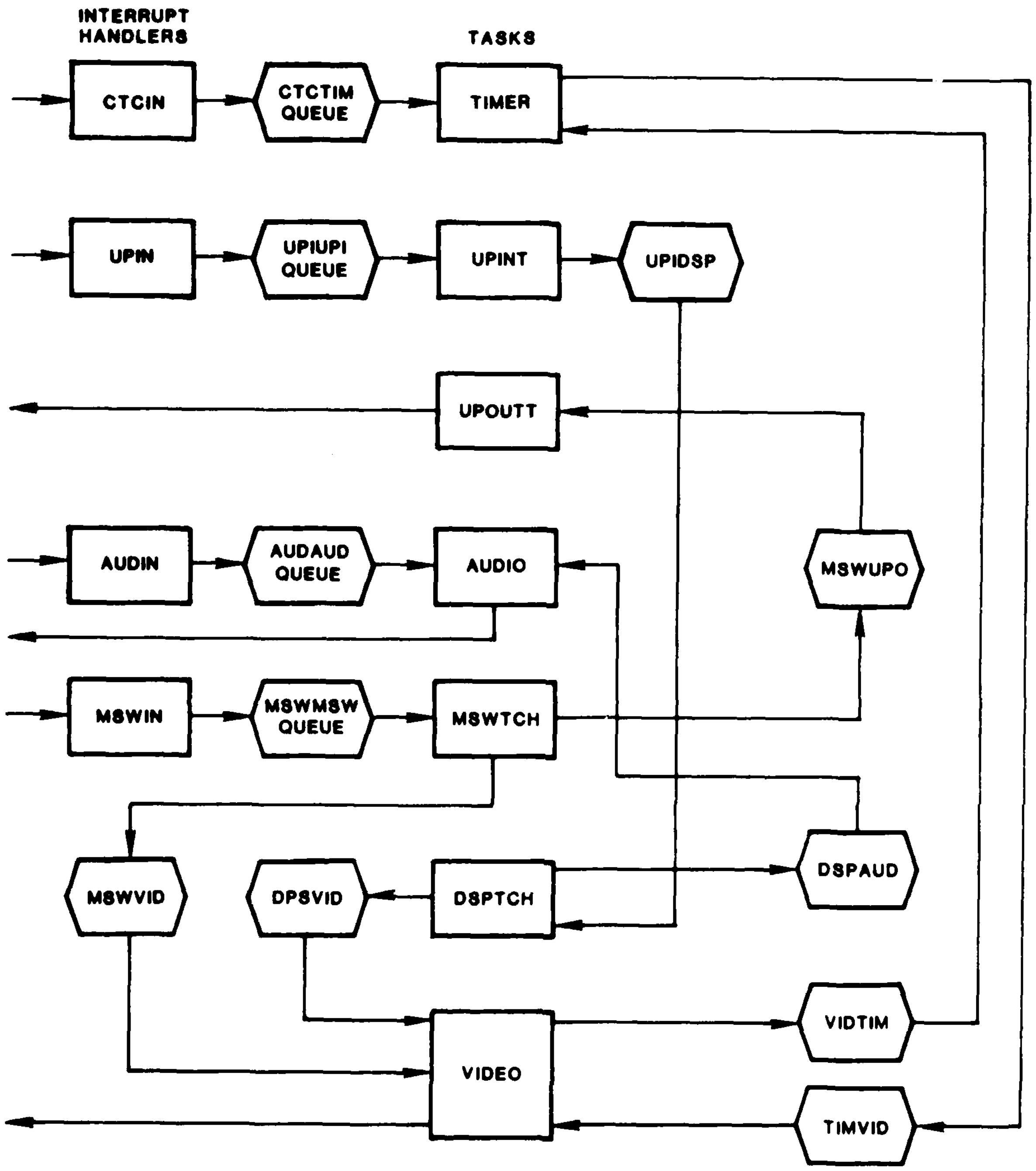


Fig. 3.3-3. Service processor data flow diagram.

3.3.2.4 The Audio Control Board Interface (audin())

The audio control board interface handler, audin(), receives an interrupt whenever the audio generator control board has completed a message. A one-byte message is queued to the audio() task to notify it, and the audio() task is awakened.

3.3.3 Tasks

The primary tasks in the service processor are shown in Fig. 3.3-1 along with their relative priority. Figure 3.3-3 shows the interaction and data flow between the tasks. The following descriptions give an overview of the primary functions of each of the tasks.

3.3.3.1 The User Processor Input Task (upint())

This task is initiated by the interrupt handler for the user processor input port. It is responsible for completing the transfer and moving the message to the command dispatch task, dsptch(). The principal reason for this architecture is to allow multiple-user processors to communicate with the service processor in an orderly fashion. There will be a separate task for each user processor on the buss.

3.3.3.2 The User Processor Output Task (upoutt())

This task has several input queues, one for each of the tasks required to transmit data to the user processor. Upoutt() scans these queues and assembles a message (less than 256 bytes) which is sent to the user processor. No interrupts are involved in this transfer, so no interrupt handler is necessary. (The user processor has the hardware and software required to synchronize this transfer.)

3.3.3.3 The Command Dispatch Task (dsptch())

This task has as input the command streams coming from one or more user processors. The commands are decoded and dispatched to the appropriate processing task (audio, video, etc.) via the queue to that task. This task, then, has the responsibility of coordinating the use of one device by several users, and resolves any conflicts in a manner consistent with the device in use. Figure 3.3-4 shows a functional flow chart for this task in the case of one user processor.

3.3.3.4 The Video Task (video())

The two main functions of the video() task are: (1) set the appropriate bits (pixels) in the video RAM in order to generate a display from the user graphic commands and (2) control the video RAM. Figure 3.3-5 gives an overall functional diagram of this task.

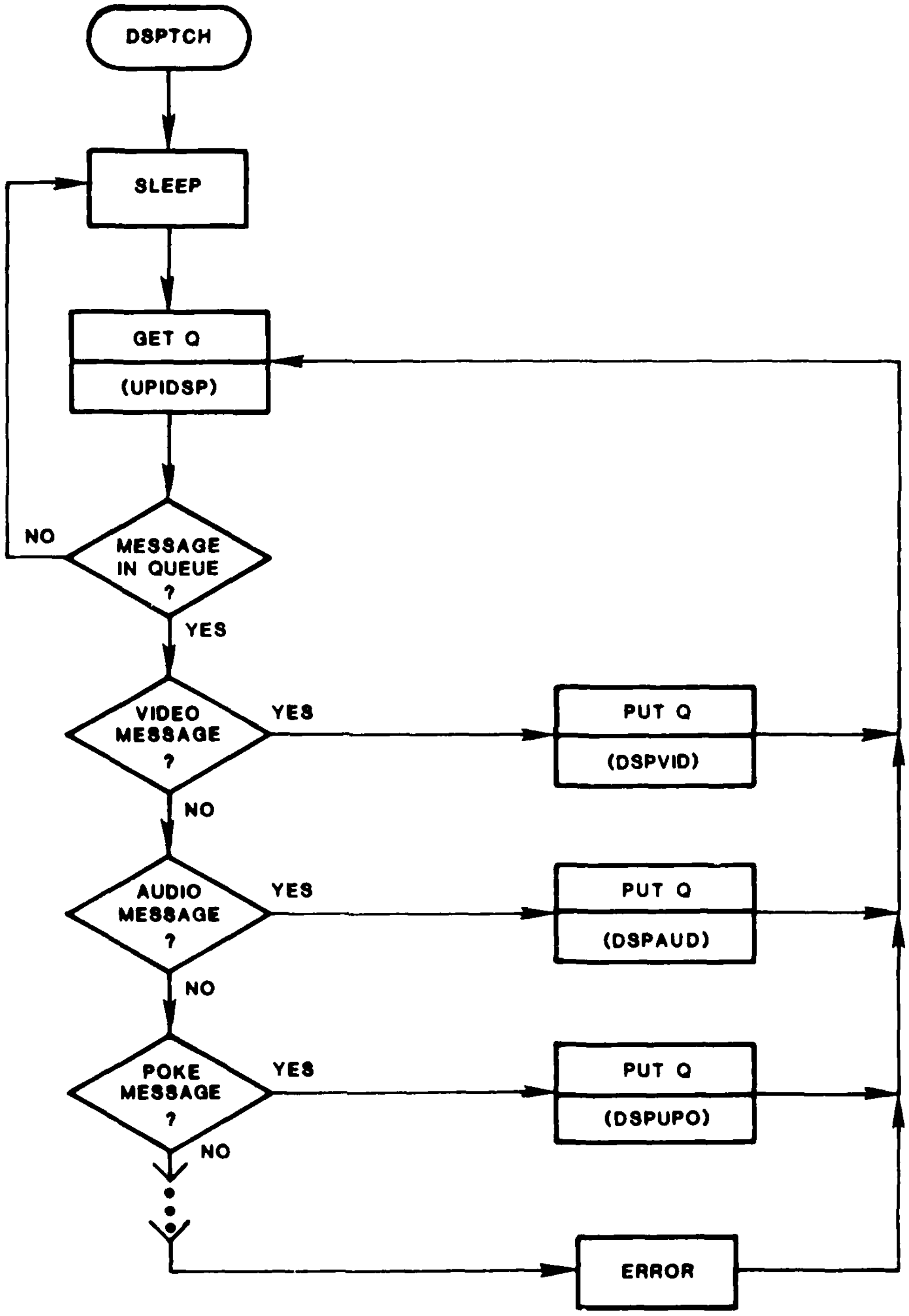


Fig. 3.3-4 Dispatch task flowchart (one user processor).

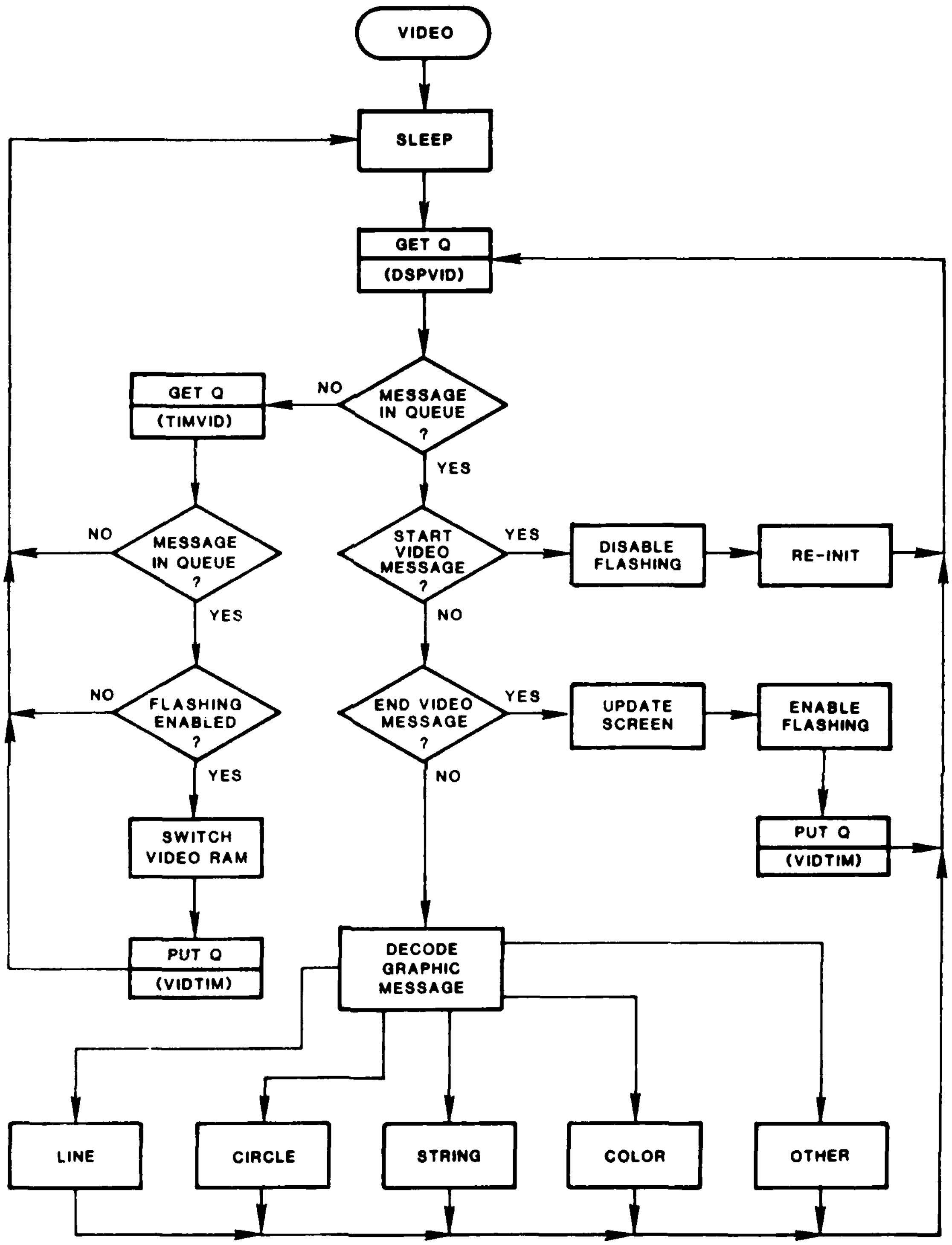


Fig. 3.3-5. Video task flowchart.

The video() task provides a set of general-purpose commands with which the user can easily generate the type of display needed. Commands provided include drawing a circle of a specified center and radius, drawing a line given two end points, and displaying a string of characters and special symbols at an arbitrary screen position. In addition, among other features, the user is able to select the color of the objects to be displayed and the type of line to be drawn (e.g., solid, dashed, dotted). Scaling commands are also provided to convert user coordinates to display coordinates so that the user need not be concerned with the resolution of the screen and other hardware specific details of the video display.

The graphics are driven on a frame-oriented basis, such that all contents of the previous frame are lost. A start-of-video message must be sent by the user to begin accumulating the graphics for the next display. As each graphic command is received, it is "drawn" into the currently available video RAM. When the end-of-video message is received, the screen is updated by switching video RAMs.

3.3.3.5 <u>The Audio Task (audio())</u>

The audio() task accepts commands of the form of an offset and length. It is assumed that the data needed to generate the audio has been loaded by the service processor into a contiguous area of memory. The audio() task uses the offset as a pointer into this area and sends the number of bytes specified by the length to the audio control card. These audio messages are accumulated until a command is received to start the audio annunciation. In this way, a number of audio messages can be stacked and then annunciated at once. (The hardware puts a limit of 4K bytes on the total that can be accumulated for subsequent annunciation).

Figure 3.3-6 gives a functional flowchart of the audio() task.

3.3.3.6 <u>The Timer Task (timer())</u>

The timer() task provides a general-purpose timing facility for other tasks. It accumulates ticks from the ctcin() interrupt routine and maintains a list of other tasks which need to be awakened after a certain number of clock ticks. The list is generated by requests coming from the other tasks and is of the form of periodic or one-shot wake-ups, both of which can be cancelled, if necessary. Because of the nature of this task, it is the highest priority task.

3.3.3.7 <u>The Mode Switch Task (mswtch())</u>

This task is awakened whenever the front panel mode switch changes state. The new state is recorded and then sent to the command dispatch task which in turn sends the setting to the user processor output task(s). The switch position controls what is displayed on the CRT: weather radar data only, AID data only, or combination weather radar/AID data.

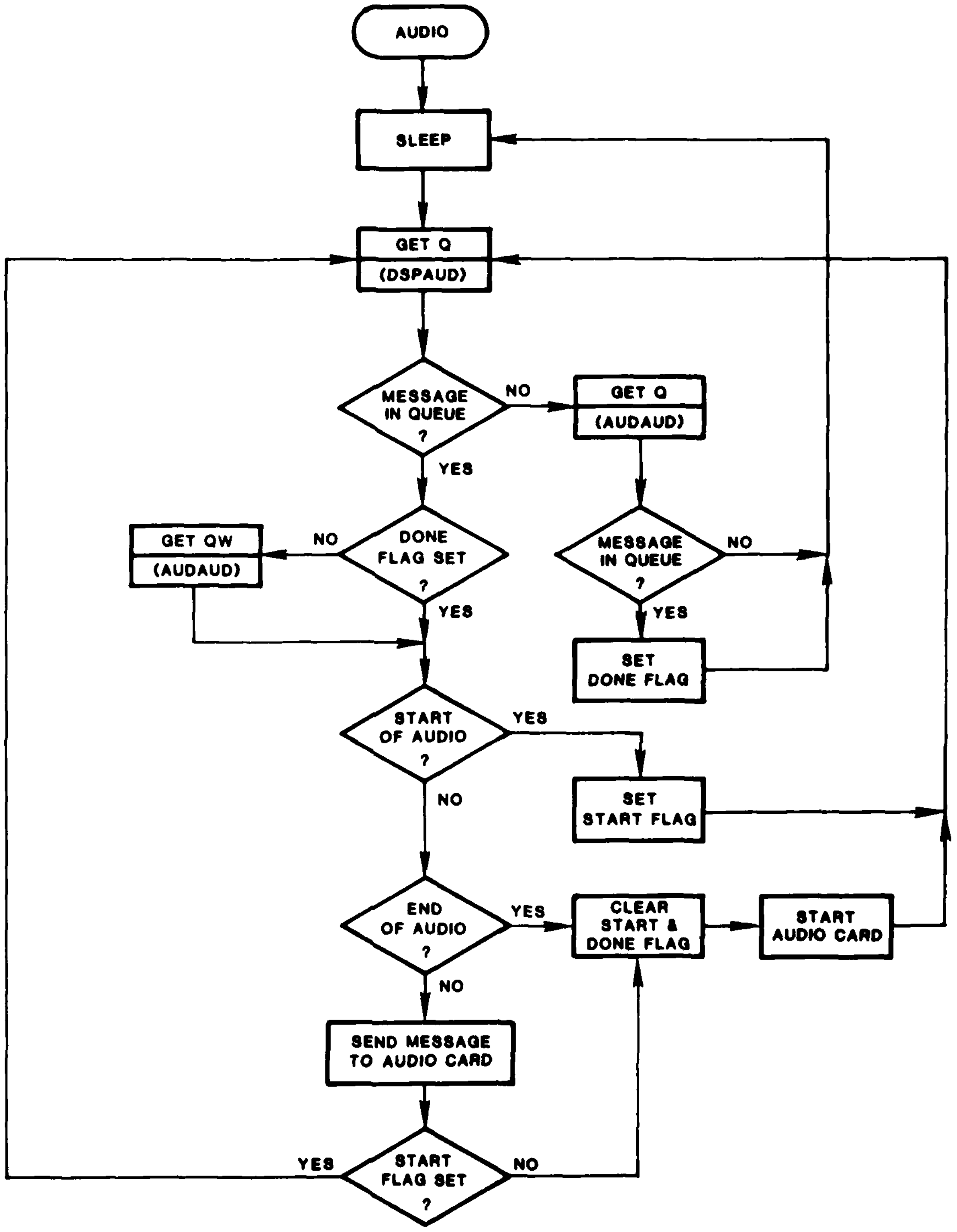


Fig. 3.3-6. Audio task flowchart.

3.4 User Processor Software General Description

3.4.1 Overview

The AID software system is designed to allow division of processing load among multiple single-board computers (SBC's) in a master/slave configuration. The master SBC, designated the service processor, serves primarily as a general-purpose audio/video processor (see Section 3.3). One or more slaves serve as user processors, each performing functions which are specific to a particular user application. I/O devices which are application-specific are attached directly to the user processor(s).

The phase I AID software provides for a single-user application and thus a single-user processor. Its function is to input aircraft information from a TCAS experimental unit (TEU) and produce audio/video output by sending appropriate graphics data blocks to the service processor.

There are five basic types of software contained in the user processor: a main program, a task scheduler, tasks, interrupt handlers, and a user graphics package. All data transferred between tasks and between interrupt handlers and tasks is passed by means of circular queues. The user processor's main program, task scheduler, and queue management protocols are similar to those in the service processor and are discussed in Section 3.2. The user processor's tasks, interrupt handlers, and user graphics package are described here.

The user processor contains six tasks and five interrupt handlers. A block diagram of these is shown in Fig. 3.4-1. The user graphics package, not shown in the block diagram, is a set of routines which may be called from any task within the user processor.

Initially all of the user processor's software is loaded from the service processor via the S-100 buss. Control is passed to the user main program, which performs a number of initialization operations, then wakes init() and calls the task scheduler. The task scheduler will immediately run init() which performs more initialization operations. Thereafter the program loops in the scheduler, continually checking for tasks which are ready to run. Tasks, once begun, may not be interrupted by other tasks, although a task may voluntarily suspend itself at any point to allow the task scheduler to schedule another higher priority task. Interrupt handlers may interrupt both tasks and other interrupt handlers depending upon priority.

There are four sources of input to the user processor: keyboard, TEU, timer and service processor. Each input has a corresponding interrupt handler: keyin(), teuin(), ctcin() and spin(). There is one output destination, the service processor, with its interrupt handler spout().

TEU inputs give position and equippage information for own aircraft and up to eight other aircraft. Data is transferred from the aircraft's TEU unit directly to the processor via an RS-232 link at one-second intervals. Own aircraft information is always included. Other aircraft information is included when available. Teuin() inputs this data block from the RS-232, places the information in the teuteu circular queue, and wakes the teu() task.

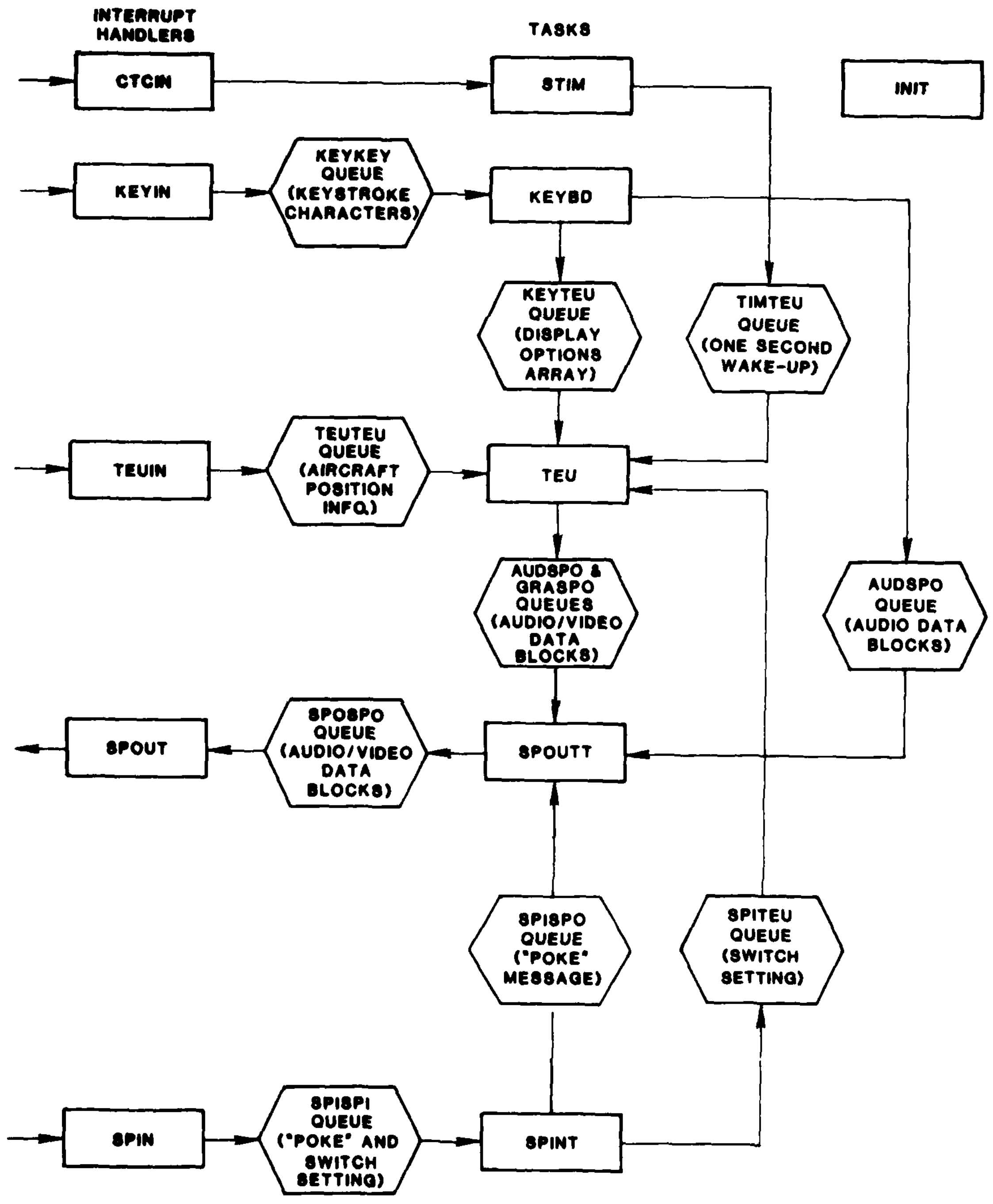

INTERRUPT HANDLERS
TASKS
CTCIN
STIM
INIT
KEYIN
KEYKEY QUEUE (KEYSTROKE CHARACTERS)
KEYBD
KEYTEU QUEUE (DISPLAY OPTIONS ARRAY)
TIMTEU QUEUE (ONE SECOND WAKE-UP)
TEUIN
TEUTEU QUEUE (AIRCRAFT POSITION INFO.)
TEU
AUDSPO & GRASPO QUEUES (AUDIO/VIDEO DATA BLOCKS)
AUDSPO QUEUE (AUDIO DATA BLOCKS)
SPOUT
SPOSPO QUEUE (AUDIO/VIDEO DATA BLOCKS)
SPOUTT
SPISPO QUEUE ("POKE" MESSAGE)
SPITEU QUEUE (SWITCH SETTING)
SPIN
SPISPI QUEUE ("POKE" AND SWITCH SETTING)
SPINT

Fig. 3.4-1. User processor functional block diagram.

The keyboard allows a user to change various TEU display characteristics, (e.g., relative or absolute altitude, maximum range displayed). Keyboard inputs consist of a single byte. They are asynchronous and may occur at any time. When a key is depressed, an interrupt is generated. Keyin() inputs the key's corresponding 8-bit byte, places it into the keykey queue, and wakes the keybd() task.

In phase I, two types of input (not including the initial program load) are received from the service processor: a 1-byte message which conveys the setting of the Bendix front panel control switch and a zero-length "POKE" message which is used to indicate that the service processor is operational. Both message types are received by spin() and passed via the spispi queue to task spint(). An acknowledgement for each is immediately sent back to the service processor via spoutt() and spout(). The switch message is passed to the TEU task where it is used in determining whether audio and video data blocks should be sent to the service processor.

Currently all user processor output is directed to the spout() interrupt handler for transfer to the service processor. With minor exceptions (see Section 3.2.2) all outputs are variable-length general-purpose graphics data blocks which have originated in the teu() or keybd() tasks and been passed through the spoutt() task to spout(). These data blocks are used by the service processor to produce audio and video output.

3.4.2 Interrupt Handlers

There are four sources of input to the user processor: keyboard, TEU, timer, and service processor, with corresponding interrupt handlers keyin(), teuin(), ctcin(), and spih(). There is one output destination, the service processor, with corresponding interrupt handler spout().

The keyboard and TEU interface to the slave via serial input ports. The slave contains a serial interface chip that supplies two serial ports (Z80-SIO). At startup seven bytes are output to each port for initialization. During program operation the interrupt handlers simply save the CPU state, input a byte and store it in a queue, then restore state and return control to the interrupted function. Keyin() wakes the keyboard task each time a byte is received; teuin() wakes the teu() task only when an entire input message has been received from the TEU.

The slave SBC contains a chip that supplies four counter-timers (Z80-CTC). In initialization three bytes are sent that set its mode, time interval and interrupt vector. During program operation, control is passed to the interrupt handler ctcin() at the selected time intervals. Ctcin() saves all CPU registers and flags on a dedicated stack, wakes the timer() task, then restores registers and flags and returns control to the interrupted function.

Communication between the service processor and user processor is via the S-100 buss. Both service processor and user processor contain two dedicated parallel ports (Z80-PIO) for S-100 buss communications. The user processor interrupt handlers spih() and spout() handle inputs from and outputs to the service-processor respectively. Spih() is awakened each time the service processor sends a byte to the user processor. The first byte of all UP-SP and SP-UP transmissions contains the byte count of the message which is to follow. Following receipt of a byte count from the SP, spih() accumulates bytes until an entire message is received. It then awakens the spint() tasks and passes the message to spint() for processing.

To initiate a transfer to the service processor, spout() outputs a single byte on the S-100 buss. This output, the byte count for the message to follow, is configured to generate an interrupt in the service processor. The service processor sets up a loop to receive the proper number of bytes. SPOUT then sends the message, this time without generating an interrupt on each byte.

3.4.3 Tasks

The user processor contains six tasks: initialization (init()), keyboard (keybd()), TEU, interval timer (stim()), service processor output (spoutt()), and service processor input (spint()). A general description of init() is given in Section 3.2.1.2. General descriptions of the remaining five tasks are given in this section. Detailed descriptions of all user processor software is given in Section 4.3.

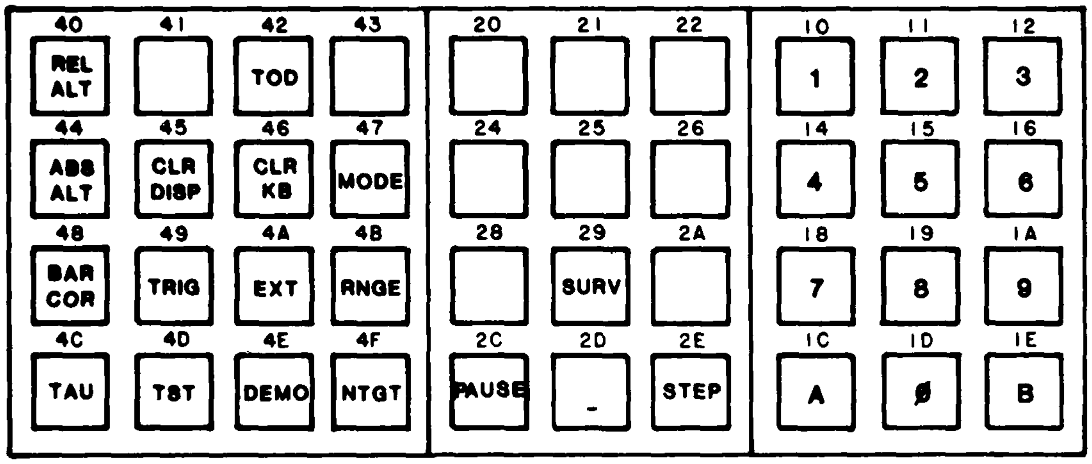
The teu() task is the major task within the user processor. With the exception of init() and stim() all other tasks serve primarily to direct data to or from the teu() task. Keyboard commands are processed by the keybd() task, then passed to teu(). Teu() uses these commands in processing aircraft position information in order to produce graphics data blocks. These data blocks are passed from teu() to spoutt() for transfer to the service processor.

3.4.3.1 The Keyboard Task (keybd())

The keyboard task has two primary functions: (1) to examine keyboard entries for validity and generate an immediate appropriate audio response and (2) to update a display options array with valid keyboard entries and send this array to the teu() task for processing. Figure 3.4-2 shows the keyboard key assignments. An overall diagram of keybd() is shown in Fig. 3.4-3.

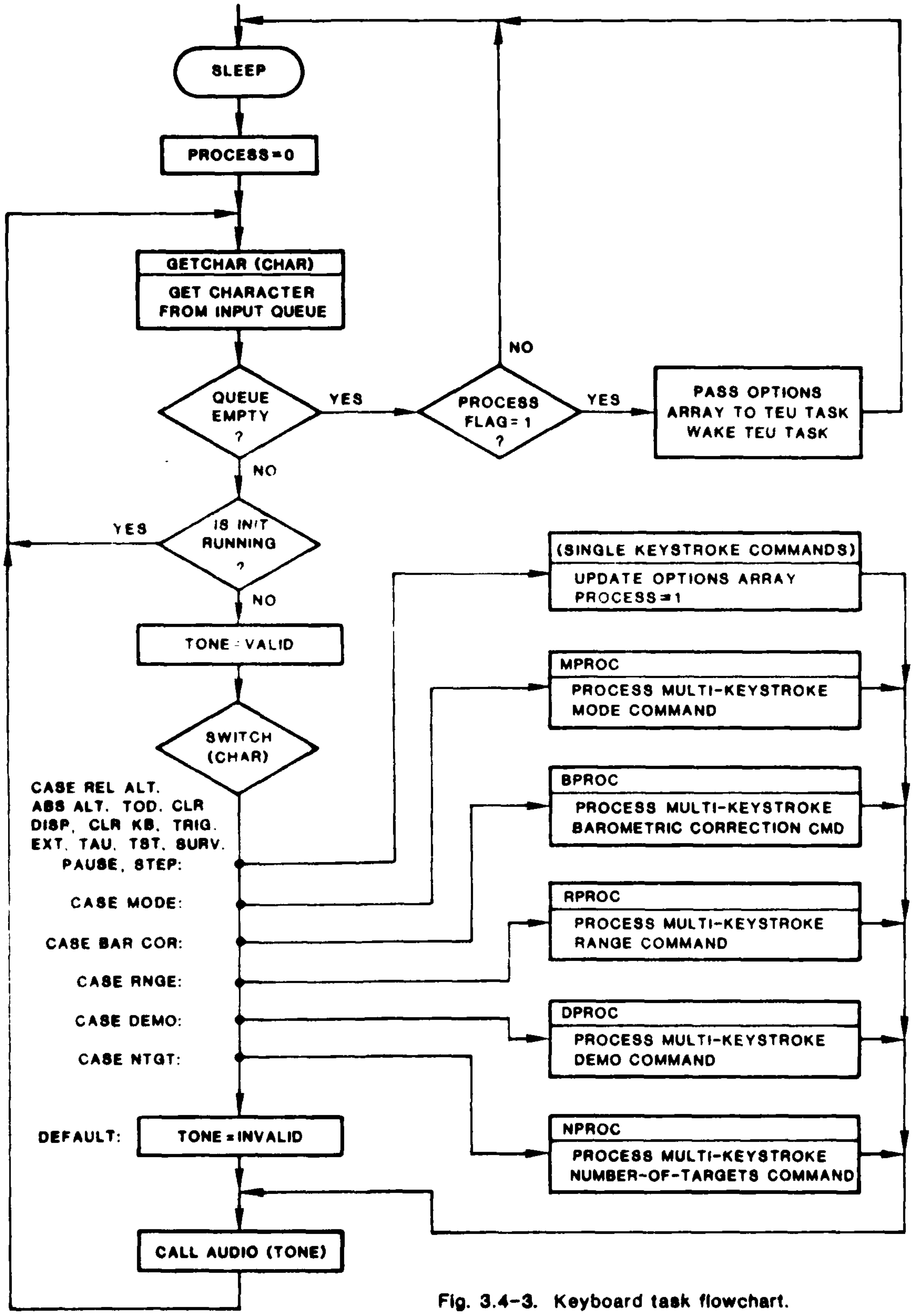
There are two basic types of user keyboard commands: those which consist of a single keystroke and those which are multi-keystroke. Keystrokes which are not properly ordered in a multi-keystroke command are considered invalid.

When keybd() is awakened, it first retrieves a character from the circular input queue. Single-keystroke valid characters result in an update of the options array and generation of a high audio tone. Single-keystroke invalid characters result in generation of a low audio tone. In either case (valid or invalid), the program then loops back to input another character.



REL ALT	Selects relative altitude format.
TOD	Places time-of-day on screen. TOD clock on board aircraft must be properly set.
ABS ALT	Selects absolute altitude format.
CLR DISP	Clears display.
CLR KB	Clears keyboard entries.
MODE ____	Selects a set of options.
BAR COR ____	Enters barometric correction in hundreds of feet. Corrections are cumulative. For use in absolute altitude mode. EXAMPLE: BAR COR - 2 would decrease own absolute altitude by 200 feet.
TRIG	Selects threat-triggered mode (proximity advisories suppressed). TRIG is default mode. Pressing key toggles between threat-triggered mode and continuous mode. In continuous mode, max range for proximity advisories is shown in green in lower right corner.
EXT	Selects extended display criteria (4 nm) for 15 seconds. "RNG 4" will appear in lower right corner of display.
RNGE ____	Selects range and autoscaling. EXAMPLE: Entering "RNGE 20" provides scale of 2 nmi to rear <u>without</u> autoscaling. Entering "RNGE 21" provides scale of 2 nmi to rear <u>with</u> autoscaling.
TAU	Selects display of current tau threshold value.
TST	Puts display in test mode.
DEMO ____	Selects canned demonstration frame. Operable only in TST mode. (00 - walking test data 01-0B fixed display 11-1A FAA scenarios)
NTGT ____	Selects maximum number of targets which will be displayed.
SURV	Selects surveillance display mode (continuous mode, 5 nm).
PAUSE	Freeze display. Operable only with FAA test scenarios.
STEP	Single-step display. Operable only with FAA test scenarios.

Fig. 3.4-2. Keyboard assignments.



SLEEP
PROCESS=0
GETCHAR (CHAR)
GET CHARACTER FROM INPUT QUEUE
QUEUE EMPTY ?
YES
NO
PROCESS FLAG=1 ?
YES
NO
PASS OPTIONS ARRAY TO TEU TASK WAKE TEU TASK
IS INIT RUNNING ?
YES
NO
TONE = VALID
SWITCH (CHAR)
CASE REL ALT. ABS ALT. TOD. CLR DISP, CLR KB. TRIG. EXT. TAU. TST. SURV. PAUSE, STEP:
CASE MODE:
CASE BAR COR:
CASE RNGE:
CASE DEMO:
CASE NTGT:
DEFAULT:
TONE = INVALID
CALL AUDIO (TONE)
(SINGLE KEYSTROKE COMMANDS)
UPDATE OPTIONS ARRAY PROCESS=1
MPROC
PROCESS MULTI-KEYSTROKE MODE COMMAND
BPROC
PROCESS MULTI-KEYSTROKE BAROMETRIC CORRECTION CMD
RPROC
PROCESS MULTI-KEYSTROKE RANGE COMMAND
DPROC
PROCESS MULTI-KEYSTROKE DEMO COMMAND
NPROC
PROCESS MULTI-KEYSTROKE NUMBER-OF-TARGETS COMMAND

Fig. 3.4-3. Keyboard task flowchart.

In contrast, each time the program recognizes the start of a multi-keystroke command, a subroutine specific to that command is entered. The program will remain within this subroutine, executing its own calls to input characters and generate audio tones, until either the correct sequence of characters or a keyboard clear has been entered. Only then is the options array updated and the subroutine exited. The program then loops back to the the beginning of keybd() to input a new character.

Each time the options array is updated, a flag (PROCESS) is set. When the keyboard task has emptied its input queue, it checks the flag setting to determine whether or not to output the options array and wake the TEU task before going to sleep.

3.4.3.2 The TEU Task (teu())

The teu() task is the major task within the user processor. Its functions are to input keyboard commands and aircraft position information, process the aircraft information according to the keyboard commands, and output audio/video graphics data blocks to be transferred to the service processor.

Inputs

Primary TEU inputs are from two sources: the TEU interrupt handler teuin() and the keyboard task keybd(). Inputs from teuin() arrive once per second. They are variable length data blocks which contain position and equippage information for own aircraft and up to eight other aircraft.

Users may enter keyboard commands at any time. It is the function of the keyboard task to reject invalid keystrokes and accept valid keystrokes in order to update a display options array. Each time this 16-byte display options array changes, keybd() passes it to the teu() task.

Outputs

Outputs from the teu() task are the audio/video data blocks described in Section 3.2.2 and Figs. 3.2-1, -2, -3, and -4.

Task Structure

The teu() task is made up of three levels of routines (Fig. 3.4-4). The main TEU processing routine tproc() (Fig. 3.4-5) is the highest level. It combines information in the display options array with aircraft information in order to generate calls to second-level routines (e.g., rev(), tod(), tau(), oalt(), rring(), tgt()). These routines in turn generate calls to third-level user graphics package routines (e.g., circle(), string(), color()). It is these user graphics routines which actually generate the graphics data blocks and output them via the graspo queue to the spoutt() task for transfer to the service processor.

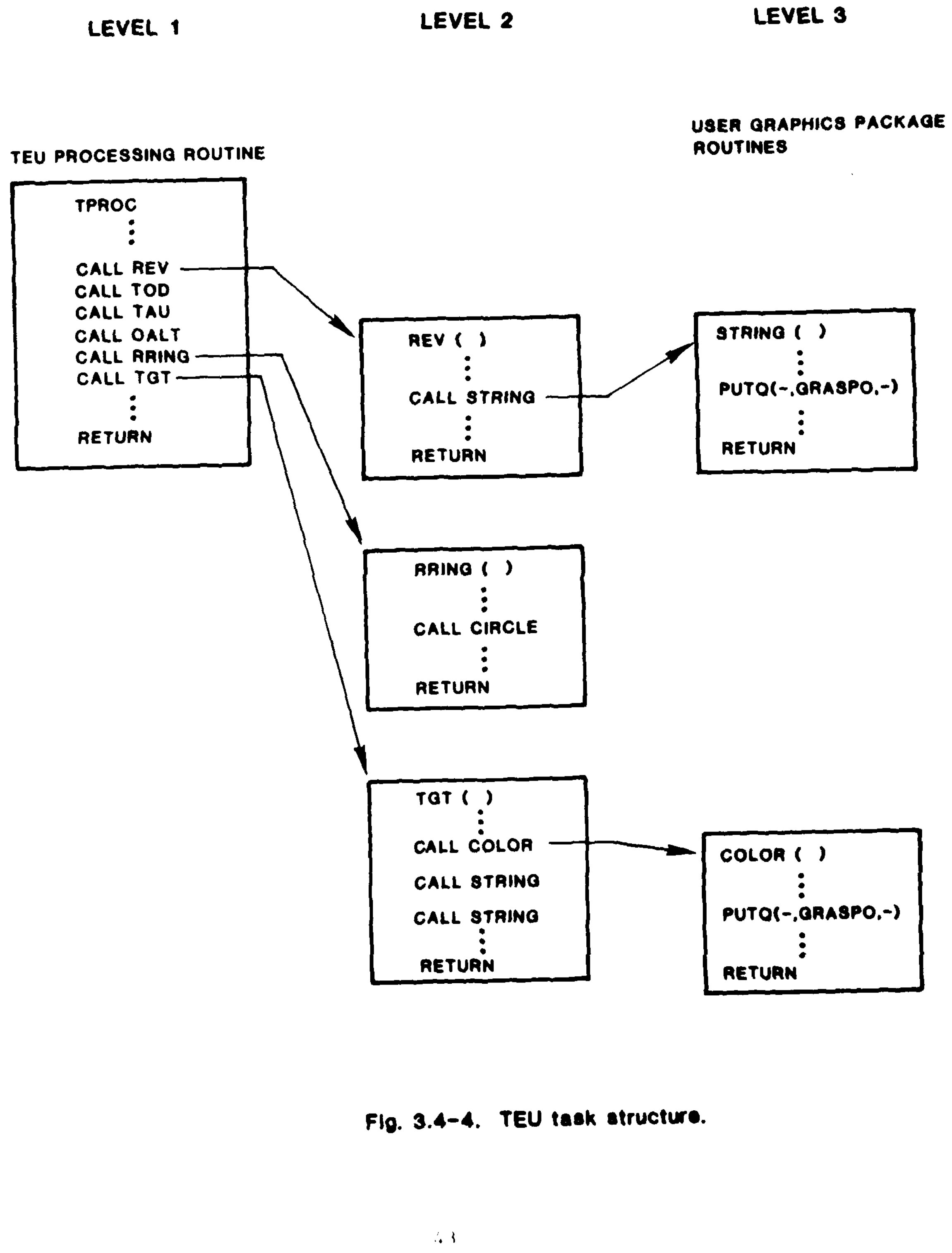


Fig. 3.4-4. TEU task structure.

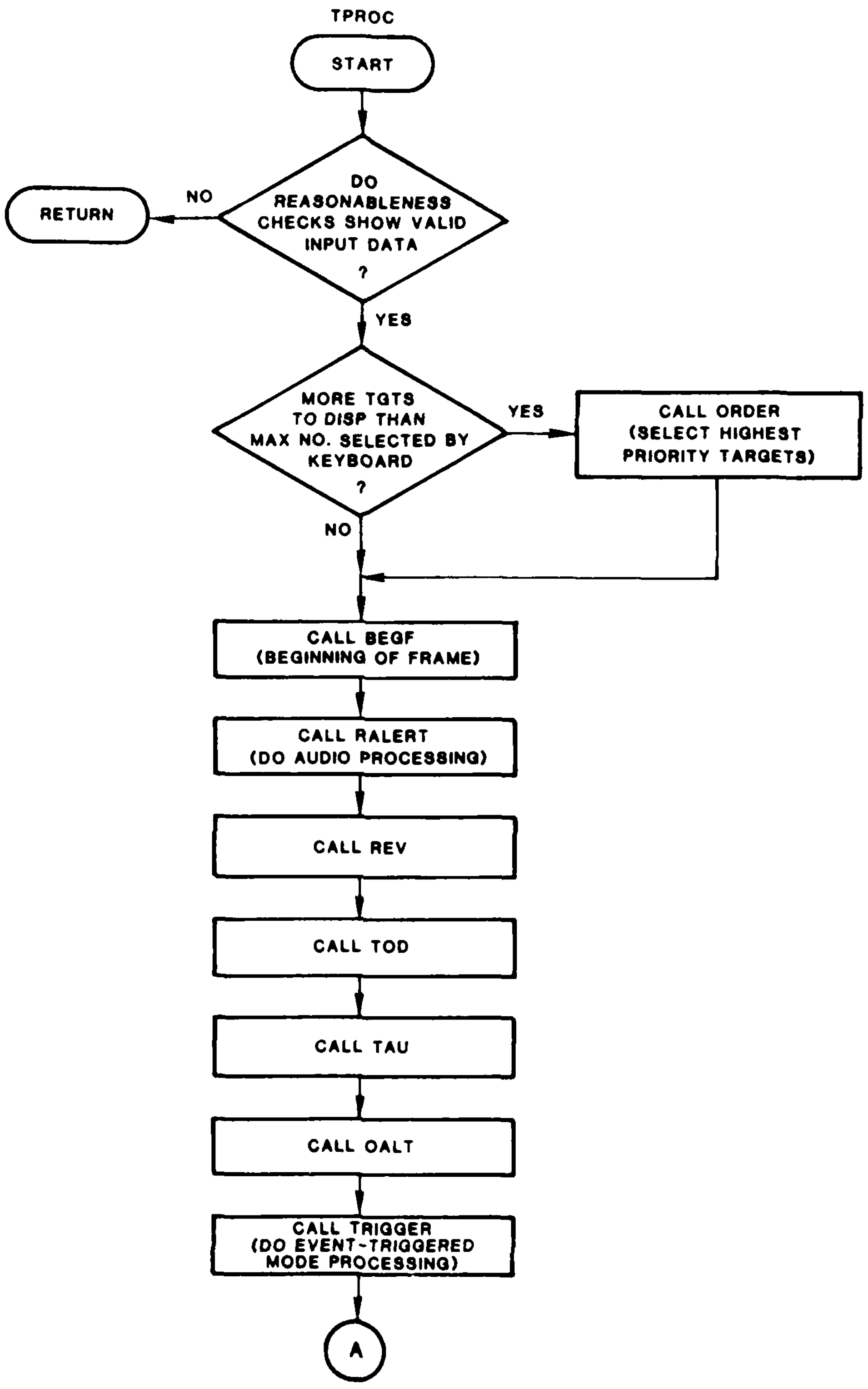

TPROC
START
DO REASONABLENESS CHECKS SHOW VALID INPUT DATA ?
NO
RETURN
YES
MORE TGTS TO DISP THAN MAX NO. SELECTED BY KEYBOARD ?
YES
CALL ORDER (SELECT HIGHEST PRIORITY TARGETS)
NO
CALL BEGF (BEGINNING OF FRAME)
CALL RALERT (DO AUDIO PROCESSING)
CALL REV
CALL TOD
CALL TAU
CALL OALT
CALL TRIGGER (DO EVENT-TRIGGERED MODE PROCESSING)
A

Fig. 3.4-5. Main TEU processing routine TPROC.

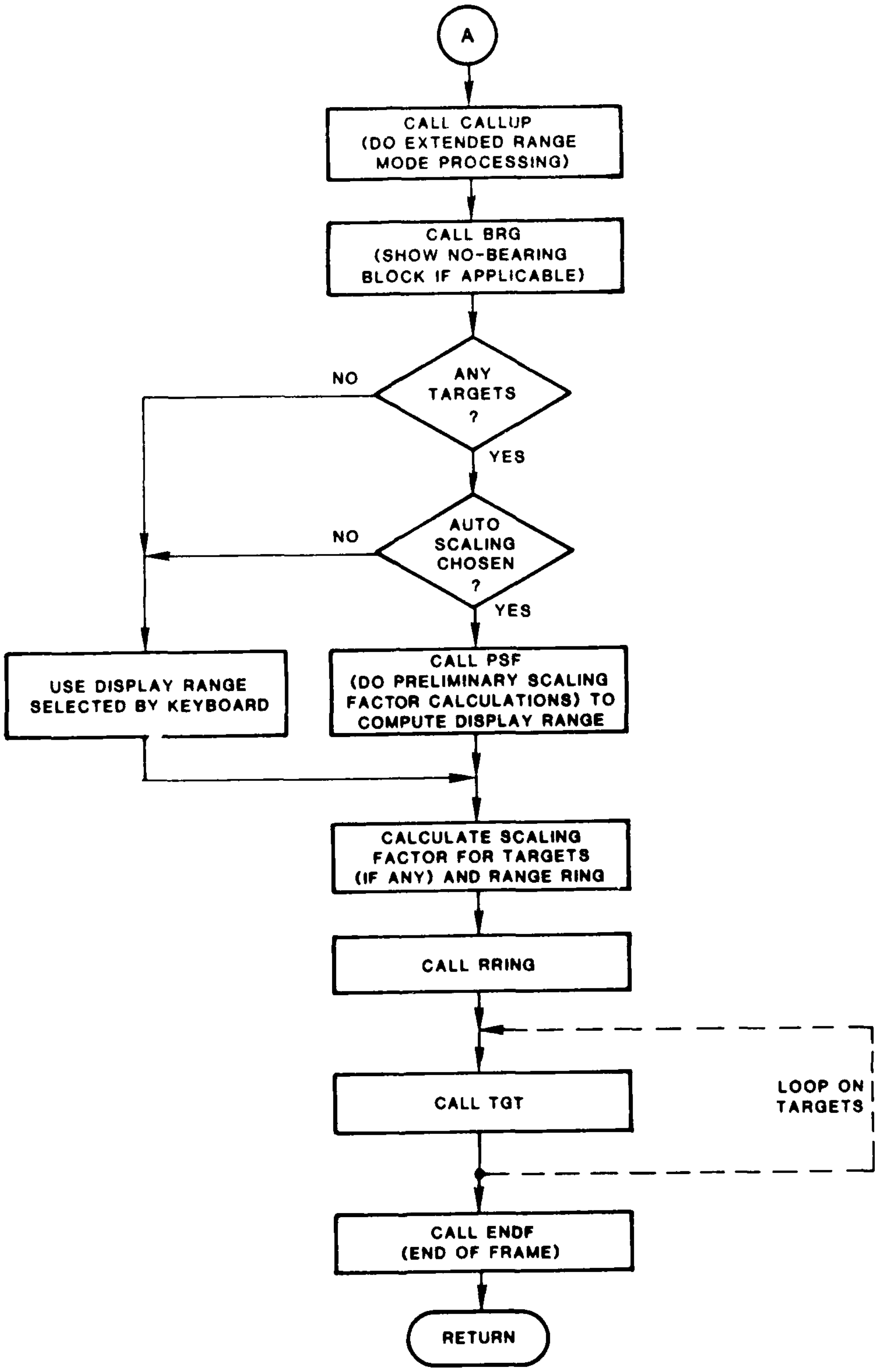

A
CALL CALLUP (DO EXTENDED RANGE MODE PROCESSING)
CALL BRG (SHOW NO-BEARING BLOCK IF APPLICABLE)
ANY TARGETS ?
NO
YES
AUTO SCALING CHOSEN ?
NO
YES
USE DISPLAY RANGE SELECTED BY KEYBOARD
CALL PSF (DO PRELIMINARY SCALING FACTOR CALCULATIONS) TO COMPUTE DISPLAY RANGE
CALCULATE SCALING FACTOR FOR TARGETS (IF ANY) AND RANGE RING
CALL RRING
CALL TGT
LOOP ON TARGETS
CALL ENDF (END OF FRAME)
RETURN

Fig. 3.4-5. Main TEU processing routine TPROC (cont'd).

Task Operation

Whenever the teu() task is awakened, it first checks its keyboard input queue. If an entry is present, this 16-byte display options array is input and used to update TEU's own display options array. Processing using this updated array is not done, however, until new aircraft inputs are received from teuin(). This means that there can be as much as a one-second lag in response to keyboard commands.

When aircraft information is received from teuin(), the main TEU processing routine tproc() is entered. Tproc() will execute once from start to finish, generating a single frame for the display. Depending upon keyboard commands and aircraft information, tproc() may execute the following routines:

ralert()	- Do resolution advisory processing. Annunciate audio. Set caution/warning lights.
rev()	- Display current software rev number in upper right corner of screen.
tod()	- Display time of day in upper left corner.
tau()	- Display current performance level's threat criteria in lower right corner. (If time to closest approach is less than threat criteria, target will be declared a threat.)
oalt()	- Display own aircraft altitude in lower left corner.
trigger()	- If threat-triggered mode has been selected, and if there are no threats or pre-threats, set parameters so that TPROC will display no targets.
callup()	- If 'call-up' or extended range mode has been selected, set display range to extended value.
brg()	- Display "no bearing" targets in block in upper left screen area.
psf()	- Do preliminary scaling factor calculations to compute minimum display range that will show all threats and pre-threats.
rring()	- Display 2-nm range ring and chevron.
tgt()	- Display target triangle and altitude tag at correct range, bearing position.

When tproc() exits, the teu() task goes to sleep to await new inputs from keybd() or teuin().

3.4.3.3 The Timer Task (stim())

The timer task provides general-purpose time interval delays to other tasks. A hardware timer is initialized to produce an interrupt every 62.5 milliseconds. Interrupt handler ctcin() then wakes stim().

When an application task wishes to start a timer it passes a count to stim() via a queue. Each time stim() runs in response to an interrupt it checks its input queues and starts new timers when entries are present. Stim() also decrements each existing counter. If the result is zero it sets the counter to -1, sends a timeout signal message to the corresponding task (via a queue) and wakes the task. If a task wishes to stop a running timer it simply sends a -1 count to stim().

3.4.3.4 The Service Processor Output Task (spoutt())

The spoutt() task receives data from three sources: "slave acknowledge" messages from the spint() task, audio data blocks from the keybd() and teu() tasks, and video data blocks from the teu() task. The function of the spoutt() task is to merge this data into a single array and pass it on to the spout() interrupt handler for transfer to the service processor. A maximum of 255 bytes may be transferred at one time to the service processor. Therefore if the combined input from the queues is more than 255 bytes, spoutt() makes more than one call to the interrupt handler, passing blocks of ≤ 255 bytes each time, until all three queues have been emptied. A call to the interrupt handler starts the transfer to the service processor. The handler then responds to interrupts to complete the transfer.

Spoutt() checks the three queues for input, reading one message from each queue in turn instead of emptying one queue before going to the next. This is done so that an audio message will not get backed up behind a long string of video messages. When all three queues have been emptied, or when the 255-byte buffer fills, whichever occurs first, the accumulated data is transferred to the spospo queue and a call is made to the spout() interrupt handler. Following this, the task either suspends itself, or if necessary, continues to read, accumulate, and transfer data until all queues are empty.

3.4.3.5 The Service Processor Input Task (spint())

The spint() task receives data from the service processor via the spih() interrupt handler. There are two types of input from the service processor to the user processor: a 1-byte message which conveys the setting of the Bendix front panel control switch and a zero-length "POKE" message which is used to indicate that the service processor is operational. When spint() is awakened, it reads the logical message passed to it by the interrupt handler. If it is a mode switch message, the message is passed on to the teu() task for use there. For all messages received, spint() sends a 2-byte acknowledgement to the service processor output task spoutt() for transmission to the service processor.

3.4.4 User Graphics Package

The user graphics package consists of a set of C-callable audio and video routines (commands). They reside in the user processor and can be called from any task within the user processor. These routines translate high-level user calls and the associated argument strings into graphics data blocks (Figs. 3.2-1 through 3.2-4) which are transferred from the user processor to the service processor for audio or video output. The graphics package acts as a software interface between the user and the display device; it frees the user from the necessity of knowing details of interface protocols and hardware configuration for a given display device. Users work with a virtual screen with units of their own choosing. A scaling command tells the service processor the number of units with which the user wishes to represent the maximum horizontal and vertical distances on the display device. Any number of audio and video commands can be grouped together using start-of-frame and end-of-frame commands to generate a single frame on the display device.

The graphics package contains four general classes of routines: audio, video control, video graphics, and miscellaneous. Descriptions of the routines contained in each class are given below. All arguments are 16-bit integers.

Audio

There are three audio commands.

CALL AUDIO (I)

where I refers to a word or phrase stored within the audio RAM. This RAM must be provided in file form by the user along with a table giving the offset and byte count for each phrase or word within the RAM.

CALL BEGA

Start-of-audio.

CALL ENDA

End-of-audio. Placing audio commands of the form CALL AUDIO (I) between BEGA and ENDA causes all of the corresponding words or phrases to be stacked in an annunicator RAM before the annunciator is activated. This allows the user to compose phrases from words that are not stored sequentially in the user's audio RAM. Audio messages not enclosed in BEGA, ENDA pairs are sent to the annunciator RAM and activated immediately.

Video Control

There are 7 video control commands. In general, a video control command selects an option which remains in effect until changed by a CLEAR command or the video control command with a different option selected. SCALE and CLEAR are special commands.

CALL SCALE (X,Y)

This routine must be called in the INIT task. It defines the coordinates of the user's virtual screen and allows the service processor to associate user coordinates with the actual physical dimensions of the display device.

CALL CLEAR

This routine clears the display and resets to the following default options: line type = solid, color = white. Any video calls following CLEAR but before the next start-of-frame call are ignored.

CALL BEGF

Start-of-frame.

CALL ENDF

End-of-frame. All commands between BEGF and ENDF are sent to the display device to be displayed as a single frame.

CALL LTYPE (I)

LTYPE selects the line type used. Current options are 0 = solid, 1 = dotted, 2 = dashed. Default is solid.

CALL COLOR (I)

Seven color options are available:

	COLOR
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	yellow
[illegible]	white

Default is white.

CALL [illegible] (I)

[illegible] turns reverse video on (I=1) or off (I=0). When reverse video is on, all characters drawn using the video graphics STRING command have their color sense reversed: pixels normally colored are now left blank; pixels normally blank are now shown in white.

Video Graphics

There are currently three video graphics commands. X and Y coordinates and circle radii used as arguments must be expressed in terms of the user coordinates selected by the SCALE command described above.

CALL CIRCLE (X,Y, radius)

CALL LINE (NSEG,XPTR)

This routine draws line segments between (X,Y) coordinate pairs (i.e., (X_1,Y_1) to (X_2,Y_2), (X_2,Y_2) to (X_3,Y_3),...,(X_{N-1},X_{N-1}) to (X_N,Y_N)).
XPTR is a pointer to an array containing the coordinates ordered X_1Y_1,X_2,Y_2,... NSEG is a signed integer. Its magnitude indicates the number of (X,Y) pairs. If NSEG is positive, the first line segment is drawn beginning at (X_1,Y_1). If NSEG is negative, the first line segment is drawn beginning at the previous cursor position. When all segments have been drawn the cursor will be positioned at (X_N,Y_N). Current software limitations allow a maximum of 9 (x,y) pairs.

CALL STRING (X, Y, NCHAR, REFPOS, CPTR)

This routine places a string of NCHAR characters on the display starting at location (X,Y). The maximum value for NCHAR is 32, limited by the width of the screen. REFPOS allows the starting X,Y coordinate to refer to various positions within the character: 0 = lower left corner 1 = upper left, 2 = upper right, 3 = lower right, 4 = center. CPTR is a pointer to an array containing the NCHAR ASCII characters.

A special feature has been provided to allow color changes within character strings. Seven 8-bit ASCII characters have been defined to represent the eight colors. In a STRING command, if a character is preceded by one of these 8-bit ASCII colors, that character (and only that one character) is displayed in the selected color.

Miscellaneous

CALL ERROR (I, TIME)

This routine allows the user processor to send an error message to the service processor for output to the line printer. I is the error number and TIME is the time (since system restart, lsb = 1 sec) at which the error occurred. An ASCII character string is generated of the form t = xxxxx, err = xxx.

CALL MODE(I)

This routine is called once per scan to tell the service processor the priority of the user-processor data. The service processor looks at this message only when the Bendix front panel mode switch is in the combination weather radar/AID position. I=1 (STANDBY) causes the service processor to ignore any AID data received from the user processor and display only weather radar data. I=3 (AID) causes the service processor to display the data received from the user processor. I is set to 3 by the TEU task when there are threats or pre-threats to be displayed or when the EXT key has been pressed on the keyboard.

4.0 SOFTWARE DETAILED DESCRIPTION

This section provides a detailed description of each of the major subdivisions of the AID phase I software: system software, service processor software, and user-processor software. This section is intended to be read in conjunction with program listings. The level of detail is that needed by a person wishing to modify portions of code.

4.1 System Software

System software provides an environment within which application programs may be run. In the case of the AID, a minimal system executive has been written to perform this function. It consists of a nonpre-emptive task scheduler and a set of data queue management functions. The queues are used to pass data between application tasks and between interrupt handlers and tasks. The same system executive is used in the AID's user and service processors.

4.1.1 The Task Scheduler and Associated Functions

A description of the design and operation of the task scheduler and queue management functions is included in Appendix A. This section will describe the implementation details of the scheduler's component parts. Five functions are involved: sched(), run(), sleep(), wake() and pause(). Briefly, the scheduler provides a mechanism for executing application tasks in response to task "wakeups" by interrupt handlers and other tasks. It chooses the next task to run based on a programmer-specified task priority.

The tasks' task control blocks (TCBs) and the tasks, themselves, are initialized as part of the startup procedure in function main(). Task initialization involves calling each task function and running it to the point where it first calls sleep(). The task may perform task-specific initialization operations during this process.

4.1.1.1 The Scheduler (sched()) Function

As described in Appendix A, a task control block (TCB) is defined for each application task. When a running task suspends (calls sleep()), sched() scans the TCBs, starting with the one for the highest priority task, until it finds one for a task that has been awakened. It then initiates execution of that task by calling run(). If no application task has been awakened, sched() simply keeps scanning TCBs. As a result, when the system is idle, the program spends its time in this TCB scanning loop. The TCBs use a linked list data structure to facilitate access.

The first operation performed in sched() is to scan the TCB linked list starting at the beginning (tidxi = 0, highest priority task). Each task's status flag, "tsksta", is tested until one is found that is set. That task's signal flag is then cleared, the address of its TCB is saved in parameter "tcbadr" for use by function sleep(), and run() is called. Function run() will initiate execution for the selected task.

When the task later suspends, control is returned to the run() function which then returns to sched(). sched() then loops back to the TCB scanning operation which searches for another task to initiate.

4.1.1.2 The Task Initiation (run()) Function

This function has an initialization mode and a normal running mode. The initialization mode is run during system initialization to compute an internal return address, RUNADR, needed by the normal running mode. The initialization mode operates if parameter "runint" is set. The initialization operation, itself, resets "runint" so that subsequent calls to run() will cause it to operate in its normal running mode. All operations within run() are performed with interrupts disabled.

The first operation performed is to test "runint". If it is set, it is cleared and the address RUNADR is computed. To do this a function "getpc" is called. This function simply gets the current value of the Z80's program counter. The value obtained is the return address for the getpc call. Address RUNADR may then be computed, since it is located a fixed number of bytes below the getpc call instruction. Address RUNADR is then stored in global parameter "runadr" and the run() function returns. This completes the initialization of run().

Under normal operation (runint = 0) the run() function saves the calling function's (sched()'s) stack pointer in parameter "mainsp" and transfers control to the address contained in parameter "slpadr". slpadr is an entry point in the sleep() function. The sleep() function simply loads the Z80's SP register with the stack pointer for the selected task (its TCB is pointed to by "tcbadr") and returns to the task. When the task again calls sleep(), sleep() saves its stack pointer in the current task's TCB and transfers control to RUNADR (using parameter "runadr") in run(). The run() function then restores sched()'s stack pointer from "mainsp", enables interrupts and returns to sched(). Note that the calls to and returns from run() use the normal C function entry/exit protocol. The return address is stored on and retrieved from the scheduler's stack.

4.1.1.3 The Task Suspension (sleep()) Function

Like run(), this function has an initialization mode and a normal run mode. The initialization mode is run to compute an internal entry address, SLPADR, needed by the run() function. The initialization mode operates if parameter "slpint" is set. The initialization mode, itself, clears "slpint" so that subsequent calls will operate in the normal running mode. All machine code operations within sleep() are performed with interrupts disabled.

The first operation performed in sleep() clears the selected task's (specified by "tidx") status flag, "tsksta". Then the task's signal flag, "tsksig" is tested. If it is set, it means that an interrupt handler rescheduled the current task to run again. In this case sleep() simply clears the signal flag "tsksig", sets the status flag, "tsksta", and returns to the calling task.

If the calling task's signal flag is not set, sleep() will return to the scheduler, sched(), via a jump to address RUNADR in run(). Sleep() first tests "slpint" and, finding it zero, transfers to location LAB3. At this point sleep() must save the current task's stack pointer in its TCB. The TCB is pointed to by "tcbadr". The stack pointer is located four bytes beyond the TCB's start address. Once the stack pointer is saved, sleep() simply jumps to address RUNADR in run(). Function run() then restores sched()'s stack pointer from "[illegible]" and returns to sched().

When sched() has selected another task to run it transfers control to address SLPADR in sleep() via a call to run(). Function run() gets address SLPADR from global parameter "slpadr". In sleep(), the stack pointer for the selected task (specified by pointer "tcbadr") is read and stored in the Z80's SP register. Function sleep() then enables interrupts and returns to the selected task using the normal C function return protocol.

4.1.1.4 The Task Wakeup (wake()) Function

A task may be awakened (i.e., scheduled to be run) by an interrupt handler or a task. A task may even awaken itself (see the pause() function). A task is awakened by calling wake(tcbidx), where "tcbidx" specifies the number (index into the TCB array) of the selected task. The wake() function simply sets the task's status (tsksta) and signal (tsksig) flags and returns. The scheduler, on testing these flags, will then cause the task to run.

4.1.1.5 The Task Pause (pause()) Function

Since the scheduler is nonpre-emptive a task must voluntarily suspend itself if other higher priority tasks must be given a chance to run. A task that requires a large amount of processing time should periodically suspend itself to allow the scheduler to run other higher priority tasks. These tasks may have been awakened by interrupt handlers. This "pause" operation is performed by the pause() function.

The pause() function simply calls wake() with the task number of the currently running task as an argument (tidx). It then calls sleep(). Control is returned to the scheduler which scans task TCBs starting with the one for the highest priority task. The highest priority awakened task is then run. Note that if no higher priority tasks are awakened, the task that originally called pause() will simply continue running from the pause() function call.

4.1.2 The Data Queues and Queue Management Functions

With the exception of parameter initialization, performed by the init() task, and a few control flags, all data transferred between tasks and between interrupt handlers and tasks is passed by means of queues. Since these functions run asynchronously it is necessary to use this mechanism so that interfunction data buffers will not be overwritten, resulting in the loss of data. The queue mechanism also provides a means for controlling the flow of data through the system so that operations are performed in the proper sequence.

The queues are implemented as circular buffers and contain variable length entries. An entry that exceeds the space remaining at the end of a buffer will be wrapped. That is, part of it will fill the remaining entries in the buffer and the rest will be stored at the buffer's start. Entries are added at the queue's tail and removed at its head.

A common data structure is defined to specify all queue headers. It is specified in the symb.h file which is attached (via an "#include" statement) to each source file. It is:

```
     typedef struct {
          int head                  ;
          int tail                  ;
          int length                ;
          char task                 ;
          unsigned char   *pbuf     ;
     }QUE                           ;
```

Parameter "pbuf" is a pointer to the actual queue byte array. "tail" points to the next open byte; "head" points to the first byte of the "oldest" entry in the queue (and hence, the next entry to be removed). Thus, if "head" equals "tail" the buffer is empty. The first byte in each entry specifies the number of bytes contained in that entry (excluding itself). Parameter "lngth" specifies the total size of the actual queue byte array. Certain queue management functions are designed to suspend the calling task if a queue is full or empty. In these cases the number of the suspending task is stored in parameter "task" so that it can be reawakened later. All queue headers are initialized in the main() and init() functions by calling functions qinit1() and qinit2(), respectively.

Nine C functions have been written to manage the data queues. Two "basic" functions, putq() and getq() perform actual data entry and retieval operations, respectively. Four functions, putqwt(), putqwk(), getqwt() and getqwk(), perform higher level operations but call the basic functions to perform actual data transfers. Finally, three minor functions, getqc(), getqd() and initq() provides queue information and management operations. These functions will be described in the remainder of this section.

4.1.2.1 The putq(source, dest, count) Function

This function moves "count" bytes from the array pointed to by "source" to the queue pointed to by "dest". If not enough room exists in "dest" to store "count" bytes (plus one more for the count byte, itself) the function returns a minus one. It also returns a zero if the queue was initially empty and a one if it was partially loaded.

The first operation performed tests the "head" and "tail" pointers. If they are equal (queue initially empty) the returned value, rtnval, is set to zero; otherwise it is set to one. Then a trial tail value, trytail, is computed, based on input argument "count". It is used to determine if the new entry would overwrite a current entry. If it would, a minus-one value is returned and the function is terminated.

The next operation tests "trytail" against the queue length to see if the entry must be wrapped.

To wrap an entry a new tail pointer, "newtail", is first computed. This potential tail pointer must then be tested against the current head pointer to see if enough room exists for the entry. If not, a minus one is returned.

If room exists, part of the entry is then stored at the end of the queue byte array; the remainder is then stored at the beginning of the array. The actual byte transfers are performed by calling function mvbyt(). This function takes advantage of the Z80's fast block move instruction. After the move, the new tail value, "newtail", is stored in the queue's header before returning.

If no entry wrap operation is necessary, the new entry is stored contiguously in the queue's buffer array. If the current tail is greater than the current head, then room exists at the end of the array (remember wrap-around was ruled out by earlier tests) and the entry is simply transferred into the queue array. The queue's tail pointer is then updated and the function returns. Also, if the trial tail, "trytail", is less than the current head, then room exists inside the array and the entry is stored. However, if the trial tail equals or exceeds the current head, insufficient room exists in the queue and a minus one is returned.

4.1.2.2 The getq(source, dest) Function

This function moves a queue entry from the queue pointed to by "source" to the array pointed to by "dest". Note that it is the calling program's responsibility to insure that enough room exists at "dest" for the entry. The destination "array" may also be simply a single-byte parameter. The function returns the returned entry's byte count if an entry is present, or a minus one if the buffer is empty.

The first operation performed is to see if the buffer is empty ("head" equals "tail"). If it is, the function returns minus one. If an entry exists, the byte count is then read and the new head pointer is tested and wrapped, if necessary. The count is then used to determine if the entry was wrapped. If it was, the bytes at the end of the queue's byte array are removed, followed by the bytes at the beginning of the array. Function mvbyt() is used to make the actual byte transfer; it uses an efficient Z80 block move instruction. After the entry has been removed, the queue's head pointer is updated and the entry's byte count is returned.

If the entry was not wrapped, the entire entry can be moved by one call to mvbyt(). The head pointer is then updated and tested to see if it should be wrapped. The function then returns the entry's byte count.

4.1.2.3 The putqwt(source, dest, stask, count) Function

This function puts an entry in a queue, if room exists, or waits (i.e., suspends the task) if not enough room exists. It moves "count" bytes from the location pointed to by "source" to the array pointed to by "dest". The calling task's number is specified by "stask". This function also checks the "task" location in the queue's header to see if a task number exists. If one does, it means that an earlier getqwt() operation was performed and the queue was empty. When this occurs getqwt() loads "task" with the number of the calling task and suspends. On detecting a task number in "task", putqwt() wakes the specified task. Since putqwt() has also loaded an entry into the queue, the getqwt() operation will then be successful and the suspended task will be able to continue. Similarly, if the putqwt() function is not able to store an entry because of insufficient room in the queue, it will store its task number in "task" and suspend. Then, when getqwt() removes an entry, it will check "task" and wake the waiting task. In this way the queues are used to "gate" the flow of data through the system. A task will not run until its input queue contains data and will not finish processing an entry until room exists in its output queue to store the results.

The first operation performed is to call putq() to attempt to store the specified message in the queue. If putq() returns a zero (queue was empty) and if "task" is not empty (not minus one) then the task specified by "task" is awakened and "task" is set to empty (minus one). However, if the putq() call returned a minus one (not enough room), the calling task's number, "stask", is stored in "task" in the queue's header and the current task suspends by calling sleep(). When the task is next awakened, putq() will again be called and its returned status tested. The process will be repeated until putq() returns a value other than minus one (i.e., the message was successfully stored). The function then returns to the calling program.

4.1.2.4 The getqwt(source, dest, stask) Function

This function is the complement to putqwt(). It gets a message from the queue pointed at by "source" if the queue contains an entry. If it doesn't it stores the calling task's number, "stask", in the queue's "task" parameter and waits (suspends). If an entry is present it is moved to the array pointed to by "dest". When this function finally terminates it returns the byte count in the message received.

The first operation performed is to call getq(). If an entry exists, it will be transferred and getq() will return a number other than minus one (the byte count). If, in addition, the "task" byte is not empty (not minus one), the specified task is awakened and "task" is cleared. However, if getq() returned minus one, then the queue is empty. In this case the calling task's number, "stask", is stored in "task" and the task is suspended by a calling sleep(). When the task is next awakened, getq() will again be called and its returned status tested. The process will be repeated until getq() returns a status other than minus one (i.e., a message was successfully received). The function then returns the received message's byte count to the calling program.

4.1.2.5 The putqwk(source, dest, count) Function

This function operates similarly to putqwt() except it does not suspend if the queue does not contain enough room. Instead, like putq(), it returns minus one. However, each time it is called it checks the queue's "task" byte, and if it is set, it wakes the waiting task. As such, this function's capabilities fall somewhere between those of putq() and putqwt(). It is used to insure that if the receiving task is suspended for lack of input data, it will run as soon as its priority will allow.

The first operation performed calls putq(). Then the destination queue's "task" byte is tested. If a task number is present the specified task is awakened and "task" is cleared. When the function exits it returns the value returned from the putq() call, which may be minus one if the putq() operation was unsuccessful.

4.1.2.6 The getqwk(source, dest) Function

This function is the complement to putqwk(). It gets a message from the queue pointed to by "source" if an entry is present. If none exists it returns minus one. In addition, it checks the source queues "task" byte to see if a task suspended is awaiting storage space in the queue. If one is, it wakes the waiting task.

The first operation performed is to call getq(). If an entry is present it is transferred; otherwise getq() returns minus one. Then the function checks the queue's "task" byte. If a task number is present, the corresponding task is awakened and "task" is reset. Finally, the function returns the value returned from the getq() call. This may be either the size of the entry transferred or minus one, indicating no entry was present.

4.1.2.7 The getqc(source) Function

This function checks the next entry to be removed from the queue pointed to by "source" and returns its byte count if an entry exists or a minus one if no entry is present. The entry itself (if one exists) is undisturbed.

The first operation performed is to test to see if the queue is empty ("head" equals "tail"). If it is, the function returns a minus one. If an entry exists it reads its byte count and returns it.

4.1.2.8 The getqd(source) Function

This function simply removes an entry (gets it and dumps it) from the queue pointed to by "source" - if an entry is present. If none exists, it returns a minus one.

The first operation performed is to test to see if the queue is empty ("head" equals "tail"). If it is, the function returns minus one. If an entry is present, its byte count is read and a new trial head pointer, tryhead, is computed. It is then tested to see if it falls outside the queue array. In that case it must be wrapped and a new head pointer is computed. It is stored in "head" in the queue's header. However, if "tryhead" falls within the queue array it is used directly to update "head". Finally, if an entry was successfully dumped, the function returns a one.

4.1.2.9 The initq(source) Function

This function simply reinitializes (clears) the queue pointed to by "source". It does this by zeroing the "head" and "tail" pointers and setting "task" to minus one.

4.1.2.10 The mvbyt(source, dest, bytc) Function

This function moves "bytc" bytes from the location pointed to be "source" to the destination pointed to by "dest". It uses assembly language code and the Z80's block move instruction, LDIR, to perform the move as quickly as possible.

4.2 Service Processor Software

The service processor is intended to be a general-purpose processor in the AID system. It is the bus master and is responsible for controlling the AID display hardware, the audio annunciator system, and other utility devices. Because it is the bus master, it also has the responsibility of downloading programs to the user processor(s) during initialization.

The primary function of the service processor is control of the AID video display. A set of general-purpose commands has been provided to facilitate the generation of graphic and alphanumeric displays, thus relieving the user processor from the time required to drive the display. In addition, the general nature of the commands eliminates the user processor's need to know the detailed aspects of the display, and will thus facilitate conversion to a different type of display, should that be necessary.

In the same way, the service processor handles the audio annunciator hardware, providing a general way to select and annunciate phrases and tones.

Communication between the service processor and the user processor is via I/O ports on the S-100 bus. The protocol established for transmission is as follows: the first byte of transmission contains the count of the number of bytes to follow.

This procotol limits the number of data bytes in a single transmission to less than or equal to 255 bytes. Within each of these transmission frames are a number of logical messages of the following format: one byte specifying the message type, followed by one byte giving the length of the message, followed by the message. While in principle the logical messages could span transmission frames, it should be noted that in the current version the transmissions consist of an integral number of logical messages. The type codes and structure of the logical messages are given in MSYMB.H.

The major tasks in the service processor software are as follows:

UPINT	- User processor input task
UPOUT	- User processor output task
DSPTCH	- Message/command decode and dispatch task
VIDEO	- Video processing task
AUDIO	- Audio processing task
MSWTCH	- Mode switch processing task
TIMER	- Timer task
INIT	- Initialization task.

4.2.1 The User Processor Input Task and Associated Functions

The upint() task obtains messages from the user processor and sends them to the dsptch() task for decoding into logical messages.

4.2.1.1 The User Processor Input Interrupt Handler (upin())

By the protocol established for user/service processor data transfers, the first byte of the message is the count of the bytes in the remainder of the message. The user processor is configured so that the service processor is interrupted on this first byte only, so that it acts as a start of transmission handshake signal. The interrupt handler inputs this byte and puts it into the upiupi queue to be processed by the upint() task. It then wakes the upint() task to notify it that a transmission has started and then returns.

4.2.1.2 The User Processor Input Task (upint())

Each time the upint() task awakens, it checks the upiupi queue, which contains counts from the input interrupt handler. If a byte count is in the queue, it signifies that the user processor has started a transmission, so upint() performs an "input and repeat" operation to obtain the remainder of the message from the user processor. The use of the Z80 inir operator is possible because the user processor has been configured to assert hardware wait states if the request for an input cannot be immediately fulfilled. When the byte request is available, the wait state is released so the service processor can continue. After the entire message has been input, it is placed into the upidsp queue and the dsptch() task is awakened.

4.2.2 The User Processor Output Task (upout())

This task accepts messages from the dsptch() task via the dspupo queue and outputs them to the user processor. No interrupts are generated for the service processor in the transmission to the user processor; the service processor performs the transmission by doing an "output and repeat" operation. This is possible because the user processor has been configured to assert wait states if it is not ready to accept a byte.

Because the number of messages going to the user processor is small, the upoutt() task is configured to send one logical message per transmission. In addition, due to several constraints in the user processor receiving software, the service processor waits for each message to be acknowledged before sending the next. The acknowledgement flag (sack) is set in the dsptch() task whenever an acknowledgement message is received from the user processor.

Because of occasional problems encountered in user/service processor transmissions, an optional synchronizing byte was added to the logical message format. The sync byte is added before the message type code byte. The option can be selected by defining the symbol SYNCB in MSYMB.H, and if it is selected, it must be the same in all user and service processor I/O routines.

4.2.3 The Command Dispatch Task and Associated Functions

The primary purpose of the dsptch() task is to unpack messages from the user processor into logical messages and send these to the appropriate task for processing. It also monitors the mode switch nd sends mode switch changes to the video task and to the user processor.

Messages from the user processor arrive unmodified in the upidsp queue. These messages consist of one or more logical messages whose format consists of an optional sync byte, followed by a byte specifying the message type, followed by a byte which contains the number of bytes in the remainder of the logical message. Although the output routine in the user processor at present sends an integral number of logical messages in a single transmission, the dsptch() task has been written to allow for logical messages which span transmission boundaries.

4.2.3.1 The Dispatch Task (dsptch())

As in all tasks, the processing in dsptch() is performed in an infinite loop. The first thing dsptch() does is to check if the mode switch has changed by checking the mswdsp queue. If the mode switch has changed, it sends a message to the video task and to the user processor output task and wakes these tasks.

Dsptch() then checks for incoming messages from the user processor. If none are available, it suspends itself and when awakened, it starts again at the beginning of its outer loop. If there is a message from the user processor, it begins extracting the logical messages.

If the sync byte option has been selected, dsptch() scans the incoming message until it encounters a sync byte. It then checks the following byte for a legal message type code. If everything is okay, it obtains the length of the message and sends the message to the appropriate task, depending on the type code.

If a partial packet is encountered, it is moved to the top of the input buffer and the next message from the user processor is read in at the end of the partial packet, thus concatenating the incoming messages.

4.2.4 The Video Task and Associated Functions

The function of the video() task is to control the AID video display in response to requests from the user processor. The requests are expected to be on a frame by frame basis. That is, a frame starts with a begin-video command, followed by any number of graphics commands, and terminates with an end-video command. Commands which do not set pixels, such as color change commands, are not required to be between a begin-video and an end-video command.

It should be noted that the current version of the video task does not support flashing. It was found the constraints of update rate, maximum number of targets, and setting/clearing pixels in software did not allow enough time to perform flashing.

4.2.4.1 The Video Task (video())

The first thing the video task does is to initialize the display. It does this by blanking the screen, erasing both video RAMs, setting the default conditions, and finally setting the video control bits to correspond appropriately to the mode switch setting.

The video task then enters an infinite loop in which video commands are received one by one from the dsptch() task and are processed appropriately. The begin-video, end-video, and clear-video commands control the switching and erasing of the video RAMs, while other graphic commands are decoded and dispatched to the proper processing subroutine. Once a complete frame has been generated in the service processor RAM, switching causes the frame to be displayed and a new frame is then started.

4.2.4.2 The draw() Subroutine

This subroutine decodes all the graphic commands and calls the appropriate subroutine. For any subroutine which sets/clears pixels, a check is made to ensure that the command is in a video frame. If not in a frame, it is ignored.

The separation of the graphics commands into a different decoding subroutine was done for historical reasons when flashing mode was allowed.

4.2.4.3 The scalex() and scaley() Subroutines

These subroutines convert user coordinates to screen coordinates in the x and y directions, respectively. Scaling from user to screen coordinates was incorporated into the graphics package to minimize the impact of using a different display with a different number of pixels and a different aspect ratio. The user is free to choose any scaling in the x and y directions with the only constraint that uxmax/uymax should correspond to the actual physical aspect ratio of the display being used.

The scaling subroutines use long integers internally to maintain accuracy. The right shift is used in place of division by 2 because it does the same thing and is much faster.

4.2.4.4 The colorg() Subroutine

This subroutine selects the color in which subsequent graphic commands will be drawn. It updates a global variable which maintains the current color and it also sets the appropriate bits in the bank select port.

4.2.4.5 The circleg() Subroutine

This is the subroutine to draw circles on the display. The required inputs are the x,y coordinate of the center of the circle and the radius. Due to the complexity and time needed to generate arbitrary circles, the current version uses 7 prestored circles and requires that the radius match one of these circles. The prestored circles are saved as offsets in the x and y directions from the center; only one quadrant of the circle is stored, since the other quadrants can be generated with appropriate changes in the sign of the offsets.

By changing the value of the parameter POFF, the number of pixels drawn can be controlled. In the present version, POFF is 2, which sets every other point in the prestored circles. By doing this, the time required to draw the circle is reduced by half, and the generated circle is quite visible and not ragged.

4.2.4.6 The lineg() Subroutine

This routine generates straight lines on the display. It uses Bressenham's algorithm (see "Principles of Interactive Computer Graphics" by Newmann and Sproull) which requires no multiplications or divisions. The inputs are the number of coordinate pairs and a pointer to the coordinate pair array. If the number of coordinate pairs is a negative number, a line is drawn from the last coordinate position of the previous call to the subroutine.

The first thing lineg() does is to check if the selected line type has changed, and if it has, it updates the saved on-count and off-count. The on-count and off-count are used by the lplot() routine to determine if a pixel should be set or not. As each coordinate is generated, lplot() sets the pixel and decrements the on-count until it goes to zero. Subsequent calls to lplot() merely decrement the off-count until it goes to zero at which time the cycle is started over. This process allowed dotted, dashed, and dimmed lines to be drawn.

Next, the coordinate pair array is converted from user to screen coordinates and the parameters are initialized for the subsequent line drawing algorithm. The line drawn is always from the "last" coordinate to the "current one". If the last coordinate from the previous call is used, the "current" is set to be the first in the current coordinate array. If the last coordinate for the previous call is not used, then the "last" coordinate is set to the first pair and the "current" is set to the second pair.

The primary loop of the lineg() routine is performed for each line segment to be drawn. The "current" and "last" coordinate pairs are updated and the line is generated by calculating the intermediate points to be set. These points are calculated in one of eight different ways depending on the slope and direction of the line segment (refer to the article mentioned above for details of the algorithm).

4.2.4.7 The setpix() Subroutine and Related Routines

This is the routine used by lineg() and circleg() to set pixels in the display memory. Although it sets/clears one bit (pixel) at a time, the routine must manipulate bytes since the display memory is accessed a byte at a time. In addition, because the colors are controlled by accessing separate memory banks at the same address, the routine may need to manipulate up to 3 bytes for every pixel access. In order to minimize the bank switching overhead, an array of three bytes is maintained in memory in which the bit manipulation is done, and the actual display memory is updated when all the bits are correctly set or cleared.

The inputs to setpix() are the display coordinate of the pixel and a flag specifying whether the pixel should be set or cleared. In the special case that the x-coordinate is -1, the routine flushes the working bytes in memory to the display memory and returns. In the normal case, the routine first converts the pixel x-y coordinate to an address offset from the beginning of the display memory. The conversion is straightforward, noting that the coordinate origin is at the lower left corner of the screen, but the display memory origin address is at the upper left corner.

If the new address offset is not the same as the current (i.e., a new byte is being addressed), then the current bytes in memory are written to the display memory and the new working bytes are obtained from the display memory.

A byte with the correct bit set is then formed and used to set or clear the appropriate bit in each of the relevant working bytes.

The routines getbyt() and stobyt() are used in conjunction with setpix() to obtain and update the bytes in the banks in the display memory.

4.2.4.8 The string() Subroutine and Related Routines

Of all the video display and graphic routines, string() and its related subroutines are the most complex. This is due primarily to the fact that characters are generated in software and can be positioned anywhere on the display. Further complexity results from allowing imbedded color commands, carriage-returns, and special graphic characters in character strings. String() takes ASCII strings, generates the appropriate pixel image in an internal buffer, and then writes the image to the display memory.

As input, a tring requires the x and y coordinate of the start of the string, the reference position parameter, the number of characters in the string and a pointer to the start of the string. The reference position parameter specifies the position of the x-y starting coordinate relative to one of five points of the first character, with the lower left corner being the default position.

After checking to make sure that the byte count is positive, the string checks for the special case for appending to the last string. If appending is desired, the current coordinates are set to the last coordinates of the previous call.

Next, it goes through a loop of all the characters in which lower case is converted to upper case and all the non-printable characters are marked. In addition, the string is checked for carriage-return and line-feed. If CR-LF is found, the string is broken into two parts, the first part of which is subsequently displayed up to the CR-LF, and the second part is displayed by a recursive call to itself at the end of the display of the first part of the message.

After checking the string, the reference position is checked and the appropriate offsets from the default are set. Then the coordinates of the beginning and end of the string are checked in both the x and y directions, and the string is truncated if it exceeds the screen boundaries.

The call to setups() initializes the internal working buffer to start building the characters in the string; it also computes the starting offsets into the display memory where the completed string will be stored.

The next two loops do the actual character drawing. The outer loop goes by each character (x direction) and the inner loop draws each character in the vertical (y direction). The call to putcbt() is the place where the correct bits are extracted from the stored characters and put into the working array (see below for a detailed description). At the end of the vertical loop, the x position is updated and the next character is drawn. At the end of the horizontal (character) loop the character counts are cleared to end the "while" loop. The call to setbyt() with the first argument of minus one flushes the current working string buffer. At this point, if the string has been broken, the string calls itself with a new pointer and a new byte count to finish the second part of the message and reforms when it is drawn.

Putcbt() extracts a byte from the character dots array, rotates it if it is necessary to align to a byte boundary and stores it into the working array. It requires as input the character, the line number in the character matrix, and the x,y coordinate where the line (byte) is to be stored. The x coordinate is assumed to be at the left-hand edge of the character. If the character to be drawn is not the same as the last character drawn, putcbt() must first compute a pointer to the matrix of dots for the new character. If the high bit of the character byte is set, then it is a special graphic character; otherwise it is an ASCII character. Aptr then points to the start of the correct array of character matrices and the offset gives the index of the start of the matrix for that character. (The character dots arrays are in the source module called ASCIIG.C). The correct byte is then extracted by using the character line number. The character line number goes from 0 to 7 and is just the offset from the lower y coordinate of the character. However, for historical reasons, the character dots array is stored from top to bottom.

If the character line byte is to be stored on a byte boundary, it is stored directly into the working array, but if it is not on a byte boundary, the byte is shifted into two bytes, and the two are stored separately.

Setbyt() is the lowest level routine used by the string related routines to store character bytes into the working string buffer, which is large enough to hold the displayed characters of a string that goes across the entire screen. It computes the x and y offsets into the working buffer from the initial coordinate positions as specified in the call to setups(). If the offsets are legal, it puts the byte into the working buffer, being careful not to overwrite previously written contents.

When the string is completely built in the working buffer, it is written into the display memory with a call to stbyts(). Each "line" of the working array is written twice into the display buffer because although characters are stored as 8 bytes high, they are written 16 pixels high. After the entire array has been written to the display memory, the "opposite" color is selected and zeros are written to the same locations to clear any previously written graphics.

4.2.5 The Mode Switch Task and Interrupt Handler

The mode switch task receives an indication that the switch position has changed from interrupt handler mswin(). It then reads the new position and sends it on to the dispatch task, dsptch(). Each time a position change interrupt is received it (re)starts a delay timer. When the timer finally times out it: 1) reconfigures the PIO to detect the next switch position and 2) sends the new switch position data to the dsptch() task. In this way the timer is used to 'debounce' the rotary switch by allowing multiple changes to occur before the final position is sent to dsptch().

4.2.5.1 The Mode Switch Task (mswtch())

After initializing some parameters, the function checks two queues (mswmsw and timmsw) to determine the source of the wakeup. If neither queue contains a message then the wakeup was initiated by the init() task. In this case the startup switch position is received (via "swtprv") and sent to the dsptch() task via queue mswdsp.

If the wakeup was from the mode switch interrupt handler, a message will be present in queue mswmsw. In this case, due to switch bounce problems, more than one message may be in the queue. As a result the queue must be cleared. Next, a timer is started. The timer is used to allow time for the mode switch to settle down before its position is read. Note that in the case of severe switch bounce problems, the timer may be started several times. Eventually it will be allowed to time out and will generate a wakeup to the mswtch() task.

If the wakeup is from the timer() task, the current mode switch setting, mswmsg, is read. The three low-order bits correspond to the three switch settings. A cleared bit in one of these three positions indicates the current switch setting. Flags representing the three bits are established to facilitate testing. The flags are then tested to determine the new setting. However, its not that simple! The switch may have been moved and then allowed to drop back to its previous position. In that case the old position and the new are the same. If this occurs, no new switch position message should be sent to the dsptch() task. When this does occur, the PIO port is configured to detect the "TCAS standby" position. That is, the operator must turn the switch back to "standby" and then advance it to the desired final position.

If the new position is not the same as the previous position a new input port mask byte is prepared to allow detection of the "other two" switch positions only. In addition, the current switch position parameter, swtpos, is set and the previous position parameter, prevsw, is updated. The new mask byte is then output to the PIO.

Finally, a final test is made to make sure the new switch position is different from the old. The new switch position is then sent to the dsptch() task. Control then returns to the beginning of the task loop and the task suspends.

4.2.5.2 The Mode Switch Interrupt Handler (mswin())

This function processes mode switch interrupts from PIO channel A. An interrupt occurs when the switch position is changed. The PIO was initialized such that its logical equation logically "ORs" the unmasked lines. A line is logically true when it is zero. Therefore, the PIO's logical operation goes from false to true on the occurrence of a zero on one of its unmasked lines. This causes the interrupt. The logical equation is reset to false by outputting a control byte (in this case, a 0X97 to port 0X85).

The mswin() interrupt handler saves the state of the interrupted function and then outputs a control byte to reset the PIO. It then outputs a mask that deactivates all input lines (0XFF to port 0X85). This blocks subsequent switch bounce interrupts when interrupts are later enabled. The mask is changed in the mode switch task when it is ready to accept new switch position change data.

After enabling interrupts, mswin() sends a "signal" message, mswsig, to the mswtch() task. The message byte contains no information; the presence of the message indicates to mswtch() that a switch change has occurred. mswtch() reads the PIO's data port to determine the new setting.

With interrupts disabled the state of the interrupted function is restored. Interrupts are then enabled and control is returned to the interrupted function.

4.2.6 The Audio Task and Interrupt Handler

The audio task receives an audio request command from the dispatch task(dsptch()), transfers the prestored, digitized audio data to the annunciator buffer (4K) and starts the annunciator. The audio data is stored in the upper 16K of the master's RAM and in the 64K audio RAM board. Audio data is stored in these areas during the initial program load sequence by the AUDM.COM, AUDA.COM, and AUDB.COM programs.

A command may request a single audio message (a word or tone) or it may be part of a concatenated string of messages (a phrase). In the latter case a "start-of-audio" message is received first, followed by one or more audio "word" messages. The sequence is terminated by an "end-of-audio" message. In this case all words are sent to the annunciator's RAM before it is commanded to start. When the annunciator is finished it issues an interrupt which is received by handler audin(). Audin() sends a message to this task via queue audaud. When the annunciator is started, a two-second timer is also started. The annunciator interrupt is used to stop the timer and reset the annunciator. If the timer times out, it means that no interrupt was received from the annunciator. In this case the annunciator is simply reset. Two seconds is more than enough time for the annunciator to output all 4K of its data.

4.2.6.1 The Audio Task (audio())

After initializing the "done" flag, the first operation performed is to check the queue from the dsptch() task (dspaud) to see if an audio request caused the task to be scheduled. If no dsptch() message is present the queue from the annunciator interrupt handler is checked. A message in this queue (audaud) indicates that the previous annunciation has been completed. The task then sets the "done" flag, stops the timer (via queue audtim) and resets the annunciator (by outputting a byte to port OPT4=0X4F).

If a message is present from the dsptch() task the "done" flag is checked to see if the previous annunciation has been completed. If not, the queues from the timer task and the annunciator interrupt handler (timaud and audaud) are checked. If the timer timed out, "done" is set, the annunciator is reset and the program proceeds to process the message from dsptch(). If the interrupt handler queue contains a byte it means that the annunciator is finished. The "done" flag is set, the annunciator is reset, the timer is turned off and the program proceeds to process the message from dsptch(). If neither queue contains a message, the program cannot proceed, so sleep() is called. When the task is again awakened the two queues are rechecked, etc.

The first step in processing the message from dsptch() is to check its type for a "begin-audio" or "end-audio" type. For the former, the "start" flag is set; for the latter, the "start" and "done" flags are cleared and the annunciator is started. In both cases the program returns to check the dspaud queue again.

If the received message is not a begin or end audio control message then it is an audio data message. The program proceeds to transfer the corresponding data from the audio RAM to the annunciator's RAM. The audio data is specified by means of an offset (audbuf.offset) and a length (audbuf.lngth). The offset is used to determine the 16K audio bank in which the data resides.

The procedure begins by deselecting the currently selected bank by selecting a nonexistent bank (bank 1). The currently selected bank was one of the video banks; it will be reselected after the audio operation is completed. The offset is then tested to determine the proper audio bank to select. The bank is selected and the offset into it is computed. The data is then moved to the annunciator's 4K RAM area.

If the audio bank selected was the upper 16K of the Master's RAM it must be deselected before the previously selected video bank (as specified by parameter "bank") is reselected.

At this point the "start" flag is checked to see if the dspaud message received was part of a concatenated string. If it was, the program returns to input the next part. If not, the "start" and "done" flags are cleared, the annunciator is started, the timer is started, and the program returns to the beginning of the task loop and suspends.

4.2.6.2 The Audio Interrupt Handler (audin())

This is the handler for the interrupt from the annunciator card. The interrupt vector was set so that it points to the DI instruction at the beginning of this function. In so doing the state of the interrupted function can be saved immediately (actually, the DI instruction is not needed since the Z80 disables interrupts automatically when an interrupt occurs).

After saving the interrupted function's state the PIO port's mask is set. Note that this code seems to be redundant or it may be that it was found to be needed to make the PIO work.

After enabling interrupts the handler sends a "signal" byte, "audint" to the audio() task via queue "audaud". This byte contains no information; the fact that a byte was sent informs audio() that the interrupt was received.

Interrupts are turned off while the state of the previously running function is restored. Control is then returned to the interrupt function via the RETI instruction. The normal C function return sequence is bypassed.

4.3 User Processor Software

The AID software system is designed to allow division of the processing load among multiple single-board computers (SBC's) in a master/slave configuration. The Master SBC, designated the service processor, serves primarily as a general-purpose audio/video processor (see Section 4.2). One or more slaves serve as user processors, each performing functions which are specific to a particular user application. I/O devices which are application-specific are attached directly to the user processor(s).

The phase I AID software described in this document provides for a single-user application and thus uses a single-user processor. This user processor interfaces to a TCAS experimental unit (TEU) and a keyboard. Its function is to input TEU aircraft position information, process the information according to keyboard commands, and generate and send data blocks to the service processor for audio and/or video output. Audio output is of two types: (1) tones to indicate whether valid or invalid keys have been pressed on the keyboard, and (2) words (e.g., climb, descend) or sounds to inform the pilot of a recommended maneuver or simply draw his attention to the display. Video output is a color PWI-type display showing targets at given ranges and bearings from own aircraft which is located near the center of the screen.

There are five basic types of software contained in the user processor: a main program, a task scheduler, tasks, interrupt handlers, and a user-graphics package. All data transferred between tasks and between interrupt handlers and tasks is passed by means of circular queues. The user-processor's task scheduler and queue management protocols are similar to those in the service processor and are discussed in 'System Software', Sections 3.2 and 4.1. The user graphics package is covered in Section 3.4.4. The user-processor's main program, tasks, and interrupt handlers are described here.

The user processor contains six tasks and five interrupt handlers. A block diagram of these, along with the connecting data queues, is shown in Fig. 3.4-1. The user graphics package, not shown in the block diagrams, is a set of routines which may be called from any task within the user processor.

Initially all of the user processor's software is loaded from the service processor via the S-100 buss. Control is passed to the user main program, which performs a number of initialization operations, then calls the task scheduler. The task scheduler will immediately run the init() task which performs more initialization operations. Thereafter the program loops in the scheduler, continually checking for tasks which are ready to run.

There are four sources of input to the user processor: keyboard, TEU, timer and service processor. Each has a corresponding task (keybd(), teu(), stim(), spint()) and interrupt handler (keyin(), teuin(), ctcin(), spih()). There is one output destination, the service processor, with task spoutt() and interrupt handler spoh(). The sixth task is init().

There are seven sections which follow to describe the user processor software. Section 4.3.1 covers the user-processor main program. Sections 4.3.2 - 4.3.7 correspond to the six user-processor tasks with their related interrupt handlers and functions.

4.3.1 The User-Processor Main Program (main())

Upon power-up, the slave single-board computer runs a boot program stored in an on-board ROM. This initializes the slave to receive a program download from the Master via the S-100 buss. After the slave program has been downloaded, control is passed to main(), the user processor main program. Main() performs a number of initialization operations, then wakes init() and calls the task scheduler. The task scheduler immediately runs init() which performs more initialization operations. Thereafter the program loops in the scheduler, continually checking for tasks which are ready to run. Neither main() nor init() run again unless the system is again powered-up.

In the AID software, the initialization operations have been divided into two parts. The idea was for main() to perform those operations necessary only at power-up and for init() to perform those operations necessary for a system restart. In reality, the partitioning of initialization is more suited to a system in which the program is stored in ROM and in which restart could be done by simply scheduling init(). In this system, with program stored in RAM, restart by running init() would not necessarily be successful. Hence we restart by rebooting the entire system from the disk, running both main() and init(). The idea of partitioning is retained, however, in case it should be desirable to store the program in ROM at a later date.

Main() begins by moving CPM's interrupt vectors from their high core locations to low core (starting at location 0). Since we do not currently use CPM or any of its interrupt vectors, this is simply a precaution in case of future software changes. Having the interrupt vectors start at location 0 ensures that we will not overwrite them by code or data.

The starting address of each interrupt handler we use is then loaded into the interrupt vector table: timer, keyboard input, teu input, service processor input, service processor output, and caution/warning switch. The interrupt handler addresses used are actually the starting addresses plus 3. This bypasses the normal C function entry sequence and allows the context of the interrupted function to be saved immediately when the interrupt occurs.

Next, the task control blocks (TCB's) are partially initialized. The task stacks are allocated space from B000 downward. Run() and sleep() are called to initialize them, and then each task is run to its first suspend point.

Qinit1() is called to initialize the length and buffer pointer fields of the circular buffers or queues. Finally, main() wakes init() and calls the task scheduler.

4.3.2 The Initialization Task (init())

Init() is awakened by the main program main() after power-up to complete the initialization begun by main(). The major portion of init() is devoted to initialization of the user processor hardware I/O devices.

Init() first disables interrupts. These remain disabled for the duration of the task. Init() completes initialization of the task control blocks (TCB's), then calls qinit2() to complete initialization of the circular buffers or queues. At this point the coordinates of the user's virtual screen are defined via the scale() function. This would normally be done in the initialization segment of the TEU task. However, scale() makes use of the graspo queue and thus must follow the queue initialization done in qinit2().

The rest of init() deals with the user-processor hardware I/O devices. First the CTC timer channels are initialized. Channels 0 and 1 are used to generate baud rates for SIO serial channels A and B, respectively. Either of two hardware configurations will be present: console I/O on channel A (9600 baud) and keyboard input on channel B (300 baud) or keyboard input on channel A and TEU input on channel B (9600 baud). The Sierra monitor assumes that a console will be connected to at least one of the serial ports and thus as a default configures the channels for 9600 baud. Therefore, when the console is on channel A, the software does not initialize either the corresponding CTC timer-counter device or serial port.

A detailed explanation of the use of the CTC timer-counter device and of the Z80 serial I/O and parallel I/O is given in the Sierra Data Sciences Technical Manual. This must be read in order for the I/O initialization to be understood.

The procedure to generate 300-baud rate for the keyboard is as follows: The user outputs bytes which set the channel for timer mode, set the prescaler P to 16, and set the down-counter time-constant TC to 52. This creates a pulse train of period = (system clock period)*P*TC = .25 μsec*16*52 = 208 μsec. A 208-μsec period is equivalent to 4807 pulses per second. This is divided by the prescaler 16 to get 4807/16 = 300 pulses per second. An important note is that the CTC timer runs off the 4-MHz system clock (period = .25 μsec) whereas the CTC counter runs off the external clock (1.8432 mHz in the slave + period = .54253 μsec).

The procedure to generate 9600 baud rate for the TEU input is as follows: The user outputs data bytes which set the channel for counter mode and set the down counter time constant to 12. This creates a pulse every tc*TC = .54253 μsec*12 = 6.5 μsec. A 6.5 μsec period is equivalent to 166,666.67 pulses per second. This is divided by 16 to get 9600 pulses per second.

Channels 2 and 3 are used together with channel-2 output wired to channel-3 input. Channel 2 is set to timer mode to produce a period of .25 μsec*16*125 = 500 μsec. Channel 3 is set to counter mode to produce a period of 500 μsec*125 = 62.5 msec. A bit is set in the channel control register to generate an interrupt each time the 62.5-msec interval elapses. This is used by the user-processor's timer interrupt handler ctcin() and timer task stim().

Next the serial I/O ports are initialized for keyboard input and teu input, again depending upon the hardware configuration. Detailed comments are given in the program listings and follow closely the Sierra Technical Manual.

Parallel I/O ports are next initialized. Channel A is configured for bit control mode to be used as the caution/warning switch interrupt port. Channel B is configured for output mode to be used as the caution/warning light output port. Channel C is configured for input mode to be used as the service-processor input port. Channel D is configured for output mode to be used as the service-processor output port. Again detailed comments are given in the listings.

After PIO initialization is complete, interrupts are again enabled, and the TEU timer is started to awaken the teu() task once per second. This is the end of init().

4.3.3 The Keyboard Task and Associated Functions

The keyboard allows a user to change various TEU display characteristics (e.g., relative or absolute altitude, maximum display range, number of targets displayed). Keyboard inputs consist of single bytes. They are asynchronous and may occur at any time. When a key is depressed, an interrupt is generated. Keyin() inputs the key's corresponding 8-bit byte, places it into the keykey queue, and wakes the keyboard task. The keyboard task then uses valid keyboard entries to update a 16-byte display options array which is passed to the teu() task for processing.

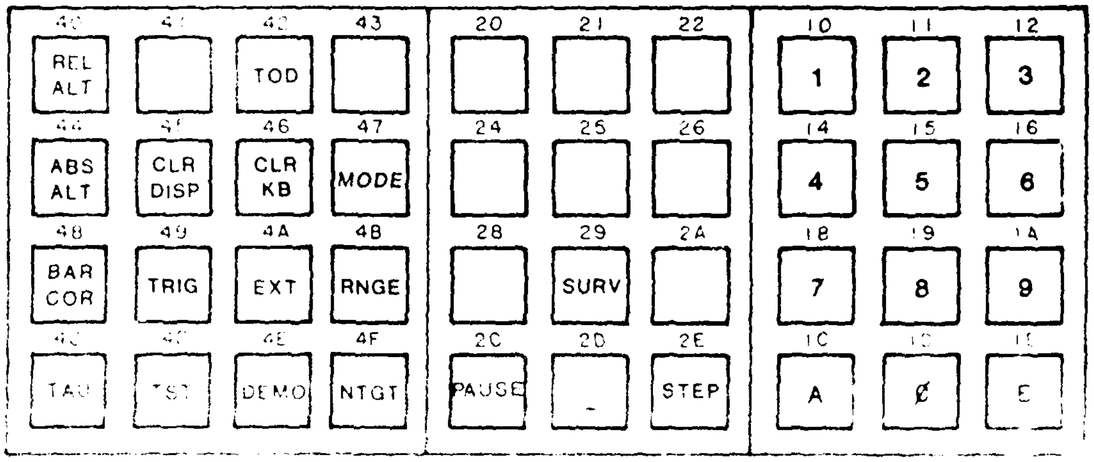
4.3.3.1 The Keyboard Interrupt Handler (keyin())

Keyboard bytes are received from the slave serial I/O, configured as either channel A (port 0X80) or channel B (port 0X82). The normal configuration is for keyboard inputs to be received on channel A and TEU inputs to be received on channel B. However, our slave single-board computer allows only two serial channels, and at times it is desirable to connect one of these to a console for debugging. In this case, console input/output is via channel A and keyboard input is via channel B. This is the reason for the conditional compile in keyin().

When a key is depressed on the keyboard, an interrupt is generated and control is passed to keyin(). Interrupts are disabled, registers are saved, and the key's corresponding 8-bit byte is read into location "inchar". If room exists in the keykey queue, the byte is placed into the keykey queue and the keybd() task is awakened. If no room exists, the byte is lost. Registers are then restored, interrupts enabled, and control is returned to the interrupted program via RETI.

4.3.3.2 The Keyboard Task (keybd())

The keyboard task has two primary functions: (1) to examine keyboard entries for validity and generate an immediate appropriate audio response, and (2) to update a display options array with valid keyboard entries and send this array to the teu() task for processing. The keyboard key assignments, along with a brief summary of keyboard commands, are shown in Fig. 3.4-2. A more detailed description of valid keyboard commands is given in Fig. 4.3-1. A description of the 16-byte display options array is given in Fig. 4.3-2.



[illegible]

[illegible] [illegible]

[illegible] [illegible]
[illegible]

[illegible] [illegible]
[illegible]

[illegible] [illegible]
[illegible]

[illegible] Se[illegible]t maximum number of targets to [illegible] (default = 8).

[illegible] [illegible]nable/[illegible] threat-filtered mode. If enabled, proximate A/C are not displayed unless a threat or pre-threat is being displayed (default = disabled).

[illegible] Reset to default display options array (see Fig. 4.3-[illegible]).

R[illegible] CONTRO[illegible]

[illegible] [illegible]st set of parameters selects display range [illegible]
2nd set of parameters selects fixed range [illegible]
autoscali[illegible]

Fig. 4.3-1. Keyboard commands.

	Note 1: When autoscaling is selected, range will be set to selected range except when autoscaling is necessary to show all threats and pre-threats.
	Note 2: Fixed ranges 2 to 8 are distances from own A/C to rear of display. Corresponding forward ranges are 4.7 to 19.2 nmiles.
EXT	Allow extended range display (4 nm, continuous mode) for 15 seconds.

ALTITUDE CONTROL

REL ALT	Set the display to relative altitude mode but do not clear any previously entered altitude correction. (Initial mode on power-up. The initial altitude correction is zero).
ABS ALT	Set the display to absolute altitude mode but do not clear any previously entered altitude correction.
BAR COR -9,-8,...-1,0,1,2,...,9	Set the display to absolute altitude mode and add a barometric correction of -900 to +900 feet to the previously entered altitude correction (i.e., barometric corrections are cumulative). This sum is then added to all absolute altitudes. BAR COR 0 is a special case which clears the barometric altitude correction. (Sets it to 0.)

TEST MODE

TST	Enable/disable test mode (default is real (non-test) TEU data). Used in combination with DEMO key.
DEMO 00,01,...,09 0A,0B 11,...,19,1A	When in test mode, selects a moving test scenario (00) a specific still-frame display (01-0B), or a moving FAA-defined encounter (11-1A).

Fig. 4.3-1. Keyboard commands (cont'd).

byte	associated keyboard key	description	value	default
0	CLR DISP	clear display	1 = clear display 0 = do not clear display	0
1	-	PPI/tabular	1 = PPI display 0 = tabular display	1
2	TOD	time-of-day	1 = display TOD 0 = do not display TOD	0
3	TAU	TAU limit for current performance level	1 = display TAU 0 = do not display TAU	0
4	REL ALT, ABS ALT	altitude	1 = relative altitude 0 = absolute altitude	1
5	BAR COR	barometric correction	-9,-8,...,-1,0,1,..,9 (each digit represents 100 ft)	0
6	RNGE	range	2,3,...,8 nmi	3
7	RNGE	auto-scaling	1 = autoscale 0 = do not autoscale	0
8	TRIG	threat-triggered mode	1 = threat-triggered mode 0 = continus mode	0
9	TST	test data	1 = use test data* 0 = use live TEU data	0
10	DEMO	selects a specific test data set*	00: 8 moving test targets 01,...,0B: still-frame displays 11,...,1A: moving FAA-defined encounters	
11	EXT	extended range display	1 = extended range 0 = normal range	0
12	NTGT	max. number of targets to display	0,1,...,8	8

*Note: Doption[10] is operational only when doption [9]=1.

Fig. 4.3-2. Display options array.

byte	associated keyboard key	description	value	default
13	PAUSE**	freeze display	1 = pause 0 = normal operation	0
14	STEP**	single-step display	1 = step 0 = normal operation	0
15	SURV	surveillance mode (5-nm continuous mode)	1 = surveillance mode 0 = other, as defined by TRIG and EXT keys	0

** PAUSE and STEP keys operational only when running in test mode with FAA-defined encounters, (i.e., doption [9]=1, doption[10]=11,...,1A)

STEP operational only when PAUSE=1.

Fig. 4.3-2. Display options array (cont'd).

Briefly, keyboard consists mainly of one large 'while' loop. Keybd() will loop, reading entries from the keykey queue and updating the display options array, until there are no more keykey entries. At that point, if any [illegible] the display options array has been changed, the [illegible]-byte array [illegible] to the [illegible] task via the keyten queue.

[illegible] detail, when keybd() is awakened, it first [illegible] the flag [illegible] to indicate that there have been no changes to the display options [illegible] [illegible], reading characters from the [illegible] [illegible] separate [illegible] for each valid character. There are two basic types of [illegible]: those which make up a single keystroke command, and those which [illegible] of a multi-keystroke command. Single keystroke valid characters [illegible] in an immediate update of the display options array and generation of a [illegible] tone. [illegible] invalid characters result in generation of [illegible] tone. [illegible] 'invalid', the program then loops [illegible] character.

[illegible] contrast, each time the [illegible] recognizes the first character of a multi-keystroke command, a [illegible] specific to that command is entered. The program will remain within this subroutine, executing its own calls to input characters and generate audio tones, until either the correct sequence of characters or a keyboard clear has been entered. Only then is the display options array updated and the subroutine exited. The program then loops back to the beginning of keybd() to input a new character.

Each time the display options array is updated, the flag "process" is set. When the keyboard task has emptied its input queue, it checks the flag setting to determine whether or not to output the display options array and wake the ten() task before going to sleep.

4.3.[illegible] Functions Called by the Keyboard Task

Five functions, all organized in the same manner, are called by the keyboard task to process multi-keystroke commands. They are:

mproc() to process the multi-keystroke mode command
rproc() to process the multi-keystroke range command
[illegible] to process the multi-keystroke barometric correction command
[illegible] to process the multi-keystroke demo command
[illegible] to process the multi-keystroke 'number of targets' command.

[illegible] function contains an outer 'infinite loop' for reading characters [illegible] the keykey queue until a valid character is encountered. [illegible] is then is [illegible] keystroke of the multi-keystroke command.) At this point, if the command is completed (if it is a two-keystroke command), the display options array is updated and control passes back to the main keybd() task. If it is a three-keystroke command, an inner 'infinite loop' is entered, again reading characters until a valid character is encountered, whereupon the array is [illegible] main keybd() task. At any point [illegible] the main keybd() task [illegible]

4.3.4 The TEU Task and Associated Functions

4.3.4.1 Overview

The teu() task is the major task within the user processor. Its functions are to input keyboard commands and aircraft position information, process the aircraft information according to the keyboard commands, and output audio/video graphics data blocks to be transferred to the service processor.

Aircraft position information is received from the aircraft's onboard TCAS experimental unit and read in via interrupt handler teuin(). When teuin() receives a complete data block, it places this data in the teuteu queue and wakes the teu() task. Teu() then is responsible for determining which targets to display, where and how the targets should be placed on the PWI-type display, what audio should be annunciated, and for communicating this information to the service processor.

4.3.4.1.1 Inputs

Primary TEU inputs are from two sources: the TEU interrupt handler teuin() and the keyboard task keybd(). Inputs from teuin() are placed once per second in the teuteu queue. These inputs are variable-length data blocks which contain position and equippage information for own aircraft and up to eight other aircraft. The format of the TEU input data blocks is shown in Fig. 4.3-3.

Users may enter keyboard commands at any time. The keyboard task rejects invalid keystrokes and accepts valid keystrokes in order to update a display options array. It is this 16-byte array (Fig. 4.3-3) which is passed to the TEU task in the keyteu queue.

There are three other inputs to the teu() task:

(1) The service processor sends a 1-byte message to the user processor each time there is a change in the Bendix front panel switch setting. This byte is placed in the spiteu queue for the TEU task and used in determining whether audio and video data blocks should be sent from the user processor to the service processor.

(2) The timer task stim() is initialized (and reinitialized each time through teu()) in order to wake the teu() task at one second intervals. This wake-up is used by teu() to decrement counters once per second and in test mode to process test data once per second.

(3) The third input is handled via global variables rather than a queue entry. Whenever the caution/warning button is pushed, the caution/warning interrupt handler cwin() zeroes the variables "cwyel" and "cwred", which are used by the TEU function ralert().

		0	1	2	3	4	5	6	7	
	1	hours of system time 0-23								
	2	minutes of system time 0-59								
	3	seconds of system time 0-59								
HEADER	4	/////////////			AUDIO			////	BRG	*
INFORMATION	5	BEU performance level 0-7								
	6	LS Byte own altitude								
	7	MS Byte LSB = 100 ft								
	8	MS Byte IVSI command								
	9	LS Byte								
	1	Priority 1-8				Window no. 1-8				
	2	Range 0-16 nm LSB = 1/16 nm								
	3	Range Rate ± 1280 kt LSB = 10 kt								target 1
TARGET	4	Rel. Alt. ± 9900 ft LSB = 100 ft								
INFORMATION	5	Azimuth 0-360° LSB = 360°/256								
	6	BB	NEW	BA	UP	DN	A/D	COLOR		
	1									
	2									
	3									target 2
	•									
	•									
	•									

|///| = spare

AUDIO	000	none
	001	'command'
	010	'clear'
	011	'alert'
	100	tone
BRG	0	BEU is not providing bearing data
	1	BEU is providing bearing data
PRIORITY	1	= highest
BB	1	= bad bearing
NEW	1	= new target
BA	1	= bad altitude
UP	1	= alt rate > 10 ft/sec
DN	1	= alt rate < -10 ft/sec
A/D	0	ATCRBS
	1	DABS
COLOR	00	white
	01	yellow
	10	red
	11	undefined

*NOTE: For certain prerecorded data sets, header words 4 & 5 have special meaning. If word 4 = OXEO, then word 5 contains a number identifying the data set which is to follow.

Fig. 4.3-3. TEU input data block format.

4.3.4.1.2 Outputs

Outputs from the teu() task are the audio/video data blocks described in Section 3.4.4.

4.3.4.1.3 Task Structure

The teu() task is made up of five levels of functions (Fig. 4.3-4). Figure 4.3-5 presents an alphabetical listing of these functions showing the file in which each function is located and giving a brief description of each function's purpose. The primary purpose of the level 1 function, task teu(), is to check each of the four input queues (spiteu, keyteu, teuteu, and timteu) for input. When TEU data is present in the teuteu queue, or in test mode, when the timteu queue indicates that test data should be processed, the level 2 function tproc() is called.

Tproc() then makes calls to 12 different level-3 functions. These level-3 functions handle either keyboard selected options (e.g., tod() -display time-of-day message in upper left screen corner) or handle some well-defined part of the processing which must be done each scan (e.g., ralert() -decide what, if any, audio should be annunciated).

Level-4 and level-5 functions are specialized subroutines used by certain level-3 functions. The lowest level functions will usually contain calls to user graphics package routines. It is the user graphics routines which actually generate the graphics data blocks and output them via the teuspo queue to the spoutt() task for transfer to the service processor.

4.3.4.1.4 Techniques for Dynamic Screen Allocation

There is one concept that requires explanation before many of the TEU functions can be understood. This is the method of dynamically allocating space on the screen whenever text messages are displayed for the first time or removed.

Aircraft position information is given in terms of range and bearing from own aircraft. It is beneficial for traffic displays to have greater range visibility in front of the aircraft (0°, up, on the screen) than behind (180°, down, on the screen). Therefore own aircraft is not located at the center of the display screen, and the available range from own aircraft to screen edge is different for different bearings. In addition, when text messages are displayed in the screen corners, or when 'no bearing blocks' are displayed, the space available for target display is reduced in certain directions (i.e., for certain bearings). Therefore, the user-processor software maintains a 256-element array (target bearing LSB = 360/256 degrees) to show current available range in each of the 256 bearing positions. This array is called dunits[]. Its units are consistent with the units selected by the user in the scale command (see Section 3.4.4) (our software sets the screen dimensions to be 1024 units horizontally and 768 units vertically.)

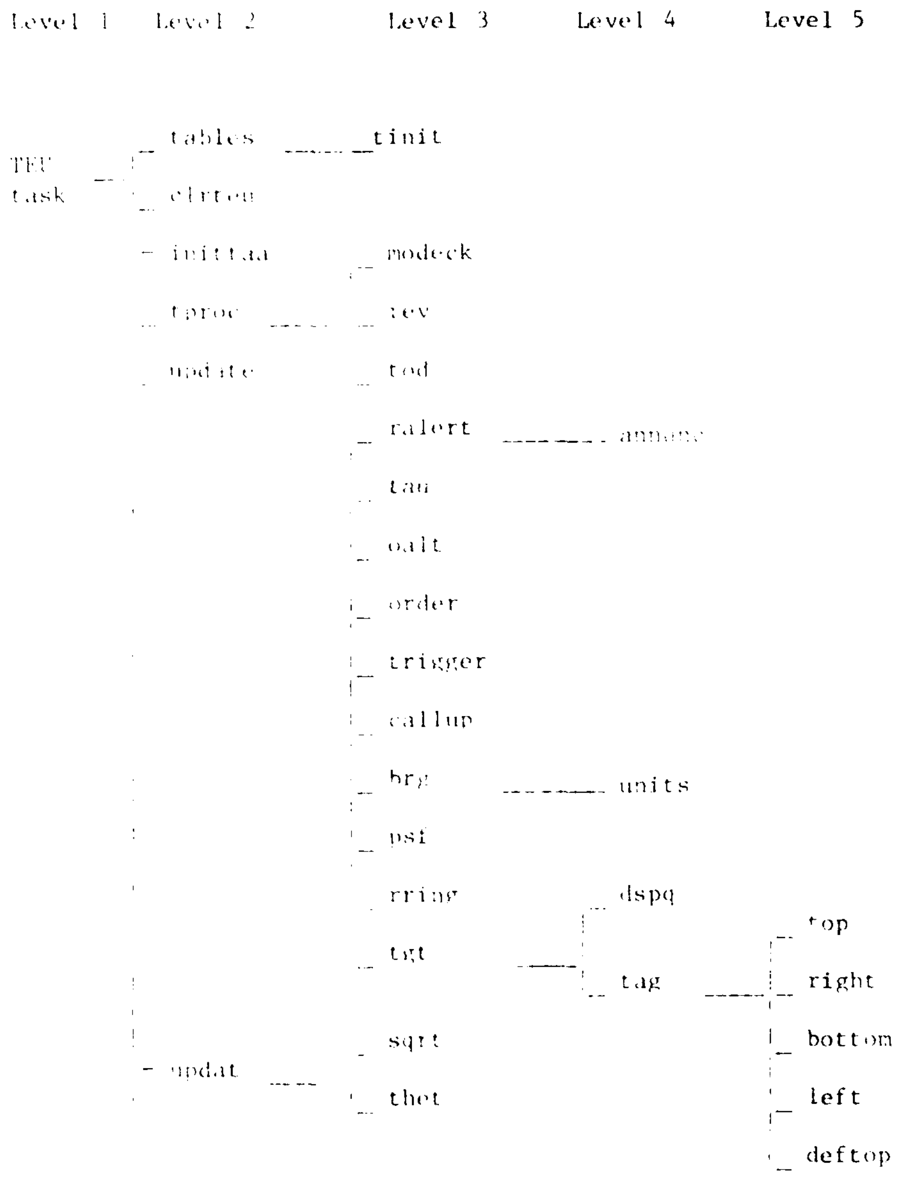


Fig. 4.3-4. TEU task structure-functions used by the TEU task.

Name	File in which function is located	Purpose
annunc	tsubs2.c	Decode IVSI command word to annunciate proper audio word.
bottom	tsubs.c	Put alt tag (if no overlap) below target triangle.
brg	tsubs1.c	Display 'no bearing' targets in block in upper left screen area.
callup	tsubs2.c	If extended range selected via keyboard figure out what display range should be.
clrteu	teu.c	(Re)initialize various screen parameters, flags, and counters. Called at start-up, when CLR DISP key is pressed, and when no data has been received for 8 seconds.
deftop	tsubs.c	No good position for alt tag. As a default, put alt tag above target triangle, even though it will overlap something.
dspw	tsubs2.c	Check to see if proximate a/c meet range criteria for display.
initfaa	faafilm.c	Called whenever a new FAA encounter is selected or a previously selected encounter repeats from the beginning. Initialize arrays and variables used in generating encounters.
left	tsubs.c	Put alt tag (if no overlap) to left of target triangle.
modeck	tsubs2.c	Set user mode switch based on mode switch setting received from Master and target severity (threat or prethreat present or ext key pressed on keyboard).
oalt	teu.c	If in absolute altitude mode, display own aircraft altitude in lower left screen corner.
order	tsubs2.c	Reorder targets in aircraft data array according to priority. Called when there are more targets received than can be displayed.

Fig. 4.3-5. Functions used by the TEU task.

Name	File in which function is located	Purpose
psf	tsubs1.c	Do preliminary scaling factor calculations to compute min. display range that will show all threats and prethreats. Called when autoscaling is selected.
ralert	tsubs2.c	Do resolution advisory processing, annunciate audio (except for commands, which are annunciated by annunc), set caution/warning lights.
rev	teu.c	Display rev message in upper right screen corner for 8 seconds after power-up.
right	tsubs.c	Put alt tag (if no overlap) to right of target triangles.
rring	teu.c	Display 2-nm range ring and chevron symbol.
sqrt	faafilm.c	Change target x,y coordinates to r, theta coordinates in updating FAA encounters.
tables	tables.c	Set up the arrays used in dynamically allocating space on the screen when text messages come and go in the corners. Called at initialization only.
tag	tsubs.c	Figure out where to put altitude tag so that it doesn't overlap target triangles or other altitude tags. Calls top, right, bottom, left, and deftop.
tau	teu.c	If selected, display the message in lower right screen corner.
tgt	tsubs1.c	Convert target position from polar to x,y coordinates. Set up information necessary to display target triangle and alt tag.
thet	faafilm.c	Change target x,y coordinates to r, theta coordinates in updating FAA encounters.
tinit	tables.c	Called at initialization only. Set up the arrays used in dynamically allocating space on the screen when text messages come and go in the corners.

Fig. 4.3-5. Functions used by the TEU task (cont'd).

Name	File in which function is located	Purpose
tod	teu.c	If selected, display time-of-day message in upper left screen corners.
top	tsubs.c	Put alt tag (if no overlap) above target triangle.
tproc	teu.c	Main TEU processing routine. Called once per scan. (See Fig. 3.4-5).
trigger	tsubs2.c	If threat-triggered mode selected via keyboard, check to see if there are any threats or pre-threats. If not, set tproc() to do no processing.
units	tsubs.c	If there is a change in the number of 'no bearing' targets displayed, make appropriate changes in arrays used in dynamically allocating space on the screen.
updat	faafilm.c	In test mode, when FAA encounters have been selected, update encounter data once per second so that targets move across the screen as specified.
update	teu.c	In test mode, (no *demonstration scenarios* or FAA encounters selected) update canned data once per scan so that targets appear to move across screen in a realistic manner.

Fig. 4.3-5. Functions used by the TEU task (cont'd)

There are nine other arrays that are used in conjunction with dunits[]: du0[], du1[],..., du8[]. Du0[] is a 256-element array which contains available units from own aircraft to screen edge for each bearing (i.e., it assumes that no text messages are being displayed and that the entire screen is available for target display). In the TEU initialization, tables() is called to set dunits[] equal to du0[].

The other arrays, du1[],..., du8[], do not contain a full set of 256 elements. For instance, du1[] contains 19 elements. When the rev message is displayed in the upper right screen corner, the 19 bearing elements in dunits[] that span the upper right screen corner will be replaced by the 19 elements of du1[]. The available ranges for those bearings will be small enough to ensure that target information does not overwrite screen text.

Much code in many of the TEU functions is devoted to changing the dunits[] array when text messages change on the screen. This is true in functions rev(), tod(), oalt(), trigger(), callup(), brg(), and units().

The numbers in du0[],..., du8[] are calculated at run time using sine and cosine tables stored in the file tables.c. The formulas used are straightforward right-triangle-type calculations, but the input numbers were derived from careful screen layout and measurement. Changes in this area would be time-consuming.

4.3.4.2 The Interrupt Handlers teuin() and cwin()

Teuin() is the interrupt handler for the TEU input interface. Bytes are received via serial I/O channel B (port 0X82). Each time a byte is received, control is passed to teuin() and the byte is read into location "inchar". Teuin() requires the TEU data to conform to an expected format: The first character of the data block must be the sync character 0XA5. The second character is the byte count of the number of bytes that follow. Teuin() will look for the sync byte, then accumulate the bytes that follow in the teuara array, 1 byte being stored each time teuin() is executed. When all bytes of a data block have been received, teuin() puts them into the teuteu queue and wakes the TEU task.

There is a timing check performed to ensure that gaps in the input data stream do not cause the data processed by the TEU task to get out of sync. When the sync character is received, the current system time (LSB= 1/16 sec) is stored in "sttim". When each subsequent byte is received, the new current system time is compared with "sttim". If the difference exceeds 3/4 second, a gap in the input data stream is assumed, the teuara array is effectively flushed, and teuin() ignores all data until another sync character is received.

There are two versions of teuin(): one version to handle TEU input with an accompanying checksum, the other version to handle TEU input without a checksum. The checksum version is located in file TEUCK.C and is the default version for use at Lincoln. The non-checksum version is located in file TEU.C and has been delivered to the FAA to interface to the Dalmo Victor TCAS unit. The checksum, when it is present, is expected to be the third byte of the TEU data, following the sync character and byte count. The data stream is considered correct when the exclusive OR of all bytes (including sync character, byte count, and checksum) yields a zero result.

Cwin() is the interrupt handler for caution/warning button inputs via parallel I/O channel A. The caution/warning button contains two separate lights. The lights can be lit separately, but the software has been set up so that pushing the button extinguishes both lights. An interrupt occurs when the button is pushed. Cwin() simply turns both lights out via an output to port 0X85 and zeroes the parameters "cwyel" and "cwred" which are used by the ralert() function. The interrupt logic is disabled in the handler to deactivate subsequent interrupts caused by switch bounce. Interrupts are re-enabled in ralert() when the caution/warning lights are turned on.

Note: The purpose of the caution/warning light/button is to direct the pilot's attention to the display when a threatening or potentially threatening situation exists. The caution/warning light/button is used in conjunction with the aural alerting logic in ralert(). When a prethreat appears, ralert() causes the yellow light to to be lit and a C-chord to be sounded. When a threat appears, ralert() causes the red light to be lit and the appropriate command to be annunciated. The lights will remain lit and the commands will be annunciated repeatedly until the pilot pushes the caution/warning button as acknowledgement.

4.3.4.3 The TEU Task (teu())

The primary purpose of teu() is to check each of the four TEU input queues (spiteu, keyteu, teuteu, and timteu) for input and direct control to the proper function for processing that input.

Teu() begins with an initialization segment which is run once at system start-up time. Tables() is called to set up the dunits[] array (see Section 4.3.4.1.4), and the used[] array is initialized (see Section 4.3.4.4.9). Initialization is also done for test mode operation. Test mode operation allows the use of either moving test data or specific fixed demonstration data sets and is explained in more detail at the end of this section.

Each time the teu() task is awakened, it checks to see if any of the four queues has input. If so, it proceeds to check each of the queues individually.

The only defined spiteu entry is a one-byte message passed from the service processor each time there is a change in the Bendix front panel switch setting. The message has one of three values corresponding to the three "on" switch settings: weather radar only, combination weather radar/AID, and AID only. The switch setting is used by the modeck() function in determining whether audio and video data blocks should be sent from the user processor to the service processor each scan. This is explained in detail in Section 4.3.6.1.

The keyteu queue is checked next. In general, the 16-byte display options array from the keyboard simply overwrites the display options array currently used by teu(). This new display options array will then be used the next time tproc() is called. Two array elements are handled in a special way. If CLR DISP has been selected, the display is cleared immediately instead of waiting for tproc() to be called. (Tproc() may not be called for some time. Under error conditions (which is usually when CLR DISP is pressed), there may be some problem getting TEU input data, and tproc() is only called when there is valid TEU input data.) Also, care is taken to ensure that the extended range or call-up mode (doption[11]) is not zeroed before it has had a chance to be processed by callup().

The teuteu queue is checked next. If the entry size is valid, the entry is read into the aircraft data array acd[]. Note: "acdl" is a one-byte field that immediately precedes acd[]. When the teuteu entry is read by getq, the entry byte count goes into acdl and the data itself (see format in Fig. 4.3-4) goes into acd[]. Tproc() is then called to process the acd[] data.

Finally, the timteu queue is checked. The timer task stim() awakens the TEU task once per second. At this time various counters are decremented. Each time new TEU data is received or each time test data is used, the restart counter "rstct" is set to RTIM. Therefore, if in operational mode no data is received for RTIM seconds, "rstct" will time out. If this happens, the display is cleared and a "no data" message is displayed on the screen.

The next section of code deals with test mode operation. Some explanation is required. The user selects test mode via the TST key on the keyboard. This sets doption[9] to 1. The TST key is used in conjunction with the DEMO key. The user presses the DEMO key followed by two digits (00,01,...,09,0A,0B,11,...,19,1A). Doption[10] is set to the value entered. If DEMO 00 is pressed, or if only the TST key is pressed without pressing the DEMO key at all, a moving test display results, with target positions being updated in a fairly realistic way each scan by update(). If DEMO 01,...,DEMO 0B is pressed, a fixed prestored demonstration scenario is displayed on the screen. If DEMO 11,..., DEMO 1A is pressed, an FAA-defined encounter is displayed, with target positions being updated each scan by updat().

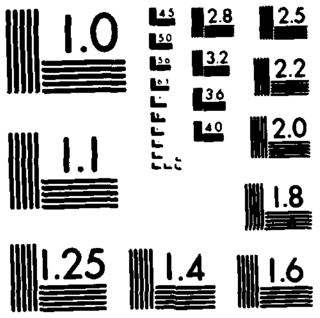
Care must be taken to initialize various parameters and arrays each time a different test display is selected. The TEU functions have some past-history memory, and without reinitialization, non-related data sets would be thought to be related. This is where the teu() parameters "canned" and "olddemo" are used. "Canned" can take on three values: 0 = real data, 1 = moving test data, 2 = fixed demonstration scenario or FAA-defined encounter. "Olddemo" is set to doption [10] showing what demo scenario, if any, was used last scan.

All data for the moving test display and the fixed demonstration scenarios is stored in the file tables.c. Teutst[] contains own aircraft header information followed by data for eight aircraft used for the moving test data. Dhdr[], demo[], dsize[], and dofts[] contain information for the fixed test data. This is well explained in the tables.c listing. All data for the FAA-defined encounters is stored in the file faafilm.c.

AD-A136 392 AIRBORNE INTELLIGENT DISPLAY (AID) PHASE I SOFTWARE DESCRIPTION(U) MASSACHUSETTS INST OF TECH LEXINGTON LINCOLN LAB A C DRUMM ET AL. 24 OCT 83 ATC-123 2/2

UNCLASSIFIED DOT/FAA/PM-83/30 F19628-80-C-0002 F/G 9/2 NL

END
DATE
FILMED
[illegible]-84
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

The teu() code checks to see if test data is being used. If so, and if no demo was selected (if moving test data is being used), and if this is the first time moving test data is being used, the aircraft data array acd[] is loaded with the moving test data. If a fixed demonstration scenario was selected, acd[] is loaded each scan with the data corresponding to the scenario, regardless of whether it was already loaded the previous scan. If an FAA-defined encounter was selected, acd[] is loaded with initial encounter data in the routine initfaa(). Whenever a demo is selected that differs from the previous demo, the used[] array is reset.

Finally, tproc() is called to process the acd[] data. Following trpoc(), if moving test data or an FAA encounter is being used, update() or updat() is called to update it.

4.3.4.4 Functions Called by the TEU Task

There are 31 functions associated with the teu() task. These are shown in Fig. 4.3-4 and Fig. 4.3-5. Only those functions which need special explanation will be covered in the sections which follow. Functions not covered below are assumed to be adequately explained by comments in the program listings.

4.3.4.4.1 The Main TEU Processing Function tproc()

Tproc() basically directs the program flow through thirteen different routines (see Fig. 4.3-4) in order to set up a single frame for the display. A brief description of each of these routines is given in Fig. 4.3-5. They handle such tasks as setting up messages displayed in the screen corners, determining what aural alerts to sound, setting up 'no bearing' blocks for targets without valid bearing information, doing the calculations connected with autoscaling, determining which targets to display and where to place the target symbols and altitude tags on the screen. There is one exception to the standard tproc() execution sequence. It occurs in playback mode when a special 'title frame' data block is sent instead of the regular TEU aircraft position information (see note in Fig. 4.3-3). In this case, a one-line title is displayed on the screen, and the normal processing initiated by tproc() is bypassed.

Pause statements appear throughout tproc() to allow higher priority tasks to run if necessary. There is a provision made in function tgt() to check to see if data from the next scan has arrived while data from this scan is still being processed. If so, the processing is lagging, and the queue which sends graphics data blocks out to the service processor is flushed. The service processor will have already received the begin frame message for this scan, but it will not receive the corresponding end frame. This causes the service processor to ignore all of the data for this scan. The effect is a temporary drop in the screen update rate from one second to two seconds. This guarantees that queues will not severely back up, which in a worst-case situation could cause the display to freeze.

4.3.4.4.2 Update()

If the user has elected to display moving test data, update() is called once per scan, immediately following tproc(), to update the aircraft position information. The intent is to cause the test targets to move somewhat realistically across the screen, with target color changing in an appropriate way and aural alerts being sounded.

When the moving test display is first selected, or whenever it is reselected following another mode of operation, tproc() initializes the aircraft data array acd[] to teutst[]. Teutst[] contains initial data values for own aircraft and eight other aircraft.

Each scan, the system time is updated. If the current time (elapsed time since the start of moving test data mode) is greater than "maxtim" seconds, acd[] is reinitialized and the sequence begins again. This reinitialization is no longer necessary and could be removed, because target movement, as currently done, could continue indefinitely.

After the time update, target information for each of the eight aircraft is updated. (Position information for own aircraft (i.e., altitude) never changes.) An 8-byte array, rarray[], is used in the updating process, each array element corresponding to one of the eight aircraft. Rarray[] is zeroed initially, signaling that the range for each target is to be decremented by "cr" each scan. Whenever an aircraft's range becomes negative, the rarray entry for that aircraft is set to 1 to cause the range to thereafter be incremented each scan. At the same time (at range cross-over), the aircraft bearing is changed by 180 degrees. This effectively causes the target to approach own aircraft from one side, pass directly above or below, and depart in the opposite direction. Whenever an aircraft's range is less than 1.0 nm, its color is set to red and an ivsi command is set. When an aircraft's range is between 1.0 and 1.5 nm, the color is set to yellow; upon transitioning from white to yellow, the aural alert "traffic" is annunciated.

4.3.4.4.3 Oalt()

Oalt() is called each scan to display own aircraft altitude if absolute altitude mode has been selected. The function is non-trivial only because the dunits array for allocating space on the screen must be updated whenever the altitude text appears for the first time or disappears. The altitude text is displayed in the lower left screen corner. In deciding how to update dunits[], it is necessary to know how many 'no bearing' blocks are being displayed on the left screen side.

The left screen side is divided into six rectangular areas (see Fig. 4.3-6(A)) These areas, when used for text, display (from top to bottom): (1) time-of-day, (2) no bearing block 1, (3) no bearing block 2, (4) no bearing block 3, (5) no bearing block 4, and (6) own aircraft altitude. The basic idea for updating dunits[] is that if two or more adjacent rectangles are available for target display, the corresponding elements of dunits[] will

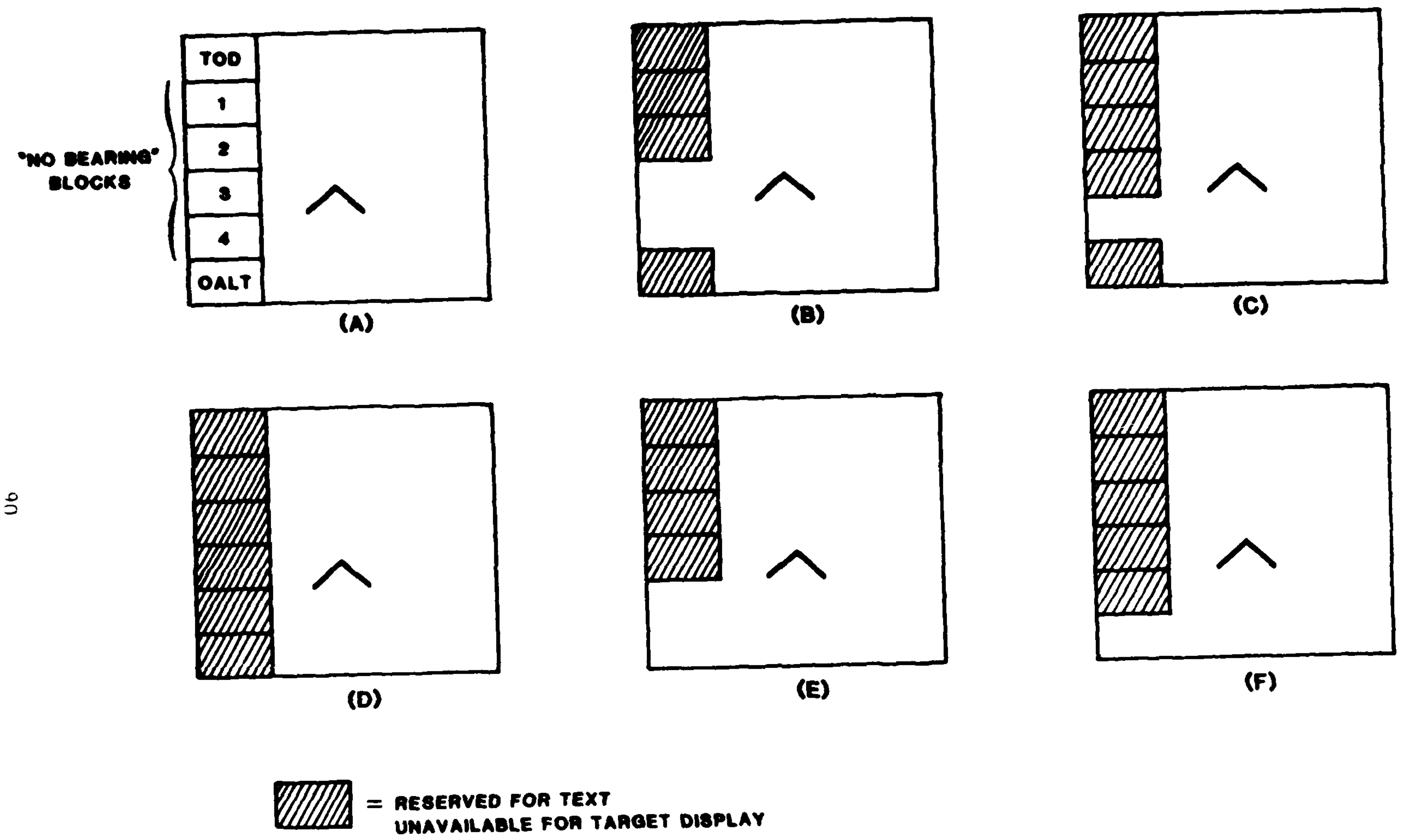


Fig. 4.3–6. Allocation of screen for text or target display.

be set for target display. However, if a single rectangle is available for target display, this is not really enough space to be of use, and the elements of dunits are set to be unavailable for target display.

The code is as follows: If own aircraft altitude is to be displayed, and if it was not displayed last scan, dunits[] must be updated to reflect a smaller display area. If 0,1, or 2 'no bearing' blocks are being displayed, simply redo the part of dunits[] that spans the lower left corner (see Fig. 4.3-6(B)). If 3 'no bearing' blocks are being displayed, only one rectangle is available for display (Fig. 4.3-6(C)). This is not enough. Redo dunits[] to show that none of the rectangles are available for target display (Fig. 4.3-6(D)). If 4 'no bearing' blocks are being displayed, the oalt block has already been declared unavailable in the brg() function, so no change to dunits[] is necessary at this time.

If altitude is not to be displayed, and if it was displayed last scan, dunits[] must be updated to reflect a larger display area. A similar procedure is followed as before. If 0, 1, or 2 'no bearing' blocks are being displayed, simply redo the part of dunits[] that spans the lower left corner. If 3 'no bearing' blocks are being displayed, free the last two rectangles on the left side (Fig. 4.3-6(E)). If 4 'no bearing' blocks are being displayed, only one rectangle is available for display (Fig. 4.3-6(F)). This is not enough. Leave dunits[] alone (i.e., leave it with all rectangles unavailable for target display as in Fig. 4.3-6(D)).

Note: A similar procedure is carried out in function units(), called when the 'no bearing' blocks are set up.

4.3.4.4.4 Order()

Order() is called whenever there are more targets available than can be displayed, as long as priority information is available to do the ordering. The TEU-associated functions always process the first "ntgt" targets in the acd array ("ntgt" = number of targets selected by the keyboard, default = 8). Therefore order() does not need to produce a priority-ordered acd[] array, only one in which all of the "ntgt" highest priority targets are within the first "ntgt" acd entries.

The approach is to first divide the acd array into two sections. When order() is finished, the first section will contain the "ntgt" highest priority targets; the second section will contain the other targets. Two pointers are used: "i", which is initialized to 0 to point to the first section, and "index", which is initialized to point to the second section. "i+9" and "index+9" are used to skip past the first 9 bytes of the acd array. These 9 bytes contain own aircraft header information, not target information.

Order() simply starts with the first acd entry and loops "ntgt" times through the acd array. Each time a target priority is less than or equal to "ntgt", it is left alone. Each time a target priority is greater than "ntgt", it switches places with the first target it comes to in the second section whose priority is less than or equal to "ntgt".

4.3.4.4.5 Trigger()

Trigger() is called to do event-triggered processing. If event-triggered mode has been selected, proximate aircraft are displayed only if a threat or prethreat is currently being displayed or has been displayed within the last TTIM (currently 8) seconds. Trigger() loops through all targets to see if there is a threat or prethreat. If so, the trigger counter "trgct" is set to TTIM. This counter is decremented once per second in teu() when an entry is received in the timteu queue. If there are no threats or prethreats and "trgct" has timed out, "tsize" is set to 0, which causes the TEU functions to process no targets.

There is a special 'surveillance' mode which takes precedence over and essentially negates trigger mode. If the SURV key has been pressed on the keyboard, all targets within 5 nm are displayed. A check is made at the beginning of trigger() to determine if surveillance mode is in effect. If so, the checks for threats and prethreats described above are bypassed.

4.3.4.4.6 Callup()

Callup() is called to do extended range or call-up processing. Extended range mode is in effect for CTIM (currently 15) seconds each time the EXT key is pressed on the keyboard. In this mode, if threat-triggered mode is also in effect, show proximate aircraft to 4 nm even if there are no threats or prethreats being displayed. (This would undo the "tsize" = 0 setting in order() above.) If continuous mode is in effect, extend the display range for proximate aircraft (show proximate aircraft to 4 nm instead of 2 nm).

The surveillance mode described in trigger() above also take precedence over callup mode. If the SURV key has been pressed on the keyboard, all targets within 5 nm are displayed. A check is made at the beginning of callup() to determine if surveillance mode is in effect. If so, callup() is not executed.

4.3.4.4.7 Units()

Units() alters dunits[] to reflect a change in the available target display area due to a change in the number of 'no bearing' blocks being displayed. Units() is called by brg(), the function which sets up 'no bearing' text blocks for targets with invalid bearing. In altering dunits[], it is necessary to also check whether the time-of-day message is being displayed in the upper left screen corner and whether the own aircraft altitude message is being displayed in the lower left screen corner. See Section 3.4.4.3, oalt(), for a similar description of how dunits[] is altered.

4.3.4.4.8 Psf()

The function psf() does preliminary scaling factor calculations, when necessary, in order to determine the screen display range, "dr".

In order to correctly position targets on the display screen, it is necessary to convert from target range, bearing units to screen x,y coordinates. The keyboard RNGE option determines in part how this conversion is done. If the auto-scaling option has been selected, psf() must first determine "dr", the range to display. If the fixed range only option has been selected, psf() simply sets "dr" equal to the selected range. In both cases, "dr" is then used to compute a scaling factor "sf" which scales all aircraft ranges for display.

Determination of Display Range

The AID software was designed to allow own aircraft to be positioned at any point on the display screen. In the current system own aircraft is centered on the screen horizontally, but located about 1/3 of the way up from the bottom vertically ((512,240) on a virtual screen of (1024,768)). When a user selects a fixed range, this range corresponds to the distance from own aircraft position to the bottom of the screen, i.e., the range at 180°, the direction of least visibility.

When the auto-scaling option is selected, a display range must be determined which allows all threats and prethreats to be visible. This is done by means of the 256-element array dunits[]. (Target bearing LSB= 360/256 degrees.) Each array element gives the number of available display units from own aircraft position to the screen edge corresponding to that bearing. Thus at 0° there are 768 - 240 = 528 available units; at 90°, 1024 - 512 = 512 available units; at 180°, 240 available units; and at 270°, 512 available units. Note that when text strings exist around the edges of the screen, the dunits[] values are reduced so that targets will not overlay the text.

The procedure to compute the display range is as follows: For each target, use target bearing as an index into the array to get available units. Divide the number of units by the target range to form the units/nm ratio necessary if the target were to lie at screen's edge. After this has been done for all targets, select the smallest ratio. This is the number of units which must equal one nm if all targets are to fit on the screen.

However, there is an additional constraint. The range selected (180° range) must be an integral number of nautical miles (2,3,4,...,8 with 2 being the smallest allowed). Therefore divide the 180° available units by the ratio (units/nm) just selected to get the range (nm) in the 180° direction. Round this up to be an integer. This is "dr", the display range to be used.

Calculating the Scaling Factor

Once the display range "dr" has been determined, or if a fixed range has been selected, or if all targets are within 2 nm (in which case "dr"=2), the scaling factor "sf" can be calculated. "sf" = 180° available units/"dr". All target ranges are then multiplied by "sf", and the radius of the 2-nm range ring is 2 * "sf".

Example

Abbreviated Array

element	bearing (degrees)	available units
0	0	528
32	45	735
64	90	512
128	180	240
192	270	512

			ratio units/nm	final radial distance in units
target 1	5 nm	45°	735/5 = 147	240
target 2	9.2 nm	0°	528/9.2 = 57.39	441
target 3	.5 nm	90°	512/.5 = 1024	24

Smallest ratio is 57.39 units/nm, i.e., 57.39 units = 1 nm for all targets to fit on the screen. Divide 180° available units by 57.39.
240/57.39 = 4.18 nm = range in 180° direction. Round up to get display range DR = 5 nm, and SF = 240/5 = 48 units/nm.
Radius 2-nm range ring is 2*SF = 96 units.
Target 1 range is then 5 nm * 48 units/nm = 240 units.
Target 2 range is 9.2 * 48 = 441 units.
Target 3 range is .5 * 48 = 24 units.

4.3.4.4.9 Tgt()

An important feature of the AID is that target altitude tags do not overwrite text or target symbols or other tags. Altitude tags may be positioned in any of four directions relative to the target triangle: top, right, bottom, or left. In addition, whenever possible, tag direction will not change from one scan to another. The result is a high level of screen clarity and readability. In order to accomplish this, in positioning altitude tags, one must keep track of the position of all previously placed target symbols and tags. This is done by means of the used array. Used[] is a two-dimensional array that stores information for up to 8 targets, with 8 fields per target: x,y coordinates of the center of the target triangle; x,y coordinates of the lower left corner of the altitude tag; new target flag; offscreen target flag; color (white, yellow or red); and most recent tag position (top, right, bottom, or left). There is also a row (the first row) for storing own aircraft chevron position information.

Target triangles are placed as accurately as possible, with overwriting of other triangles allowed. Tags, however, are positioned if possible to be in the clear.

Tgt() consists mainly of two loops through all of the targets. In the first loop, all target triangle positions are calculated and stored in the used array without regard for overlap. In the second loop, calls are made to function tag() to calculate altitude tag positions and store them in the used array. Each time tag() is called, it attempts to position an aircraft's altitude tag so as not to overlap any of the target triangles or any of the previously placed altitude tags. Tag() attempts first to place the aircraft's altitude tag in the same relative direction (top, right, bottom, left) as in the previous scan. Failing this, it trys the next direction clockwise. If all four directions fail (i.e., if the altitude tag cannot be placed in the clear), the top direction will be chosen as a default.

4.3.5 The Service-Processor Input Task and Interrupt Handler

The service-processor input task, spint(), receives two types of logical messages from the service processor via the interrupt handler: a "switch" message indicating the setting of the mode switch, and a "poke" message which is used to indicate to the user processor that the service processor is operational. In the current version, spint() expects only one logical message per transmission from the service processor and sends an acknowledgement message back for every message it receives.

4.3.5.1 The Service-Processor Input Task (spint())

The spint() task is constructed as an infinite loop. Each time it is awakened, it checks to see if a message has come from the interrupt handler via the spispi queue. If there is no message, the task merely suspends itself again. If there is a message, it clears the first byte(s) of its input buffer, and then reads the message from the spispi queue into the input buffer. (The first byte(s) are cleared to effectively remove any previous message in the input buffer.) If the message is a mode switch message, it is sent to the TEU task via the spiteu queue, and the TEU task is awakened to notify it of the receipt of the message. This is the only type of message that is currently checked.

Then, for every message received, an acknowledgement message is sent back to the service processor, via the spispo queue and the service processor output routine, spoutt(). The service processor will not send another message until the last one is acknowledged - this protocol simplifies the interrupt handler by allowing it to be singly buffered. In addition to sending the acknowledgement message, spint() also sets a flag to indicate to the output routine that a message has been received. This is only really significant on the first message from the service processor.

After processing the input message, spint() again checks the input queue for messages and continues its infinite loop.

4.3.5.2 The Service-Processor Input Interrupt Handler (spih())

The slave processor is configured to be interrupted for every byte sent to it from the service processor. For this reason, unlike the service processor, it must distinguish in software between the first byte of a message and its subsequent bytes. It does this by maintaining a flag called ioinp, which is cleared initially. Whenever an interrupt occurs, after saving the registers, the interrupt routine checks this flag and does one of two different things. If the flag is cleared (the else clause), the byte input is the first byte, which, in the transmission format established for service/user processor communication, is the byte count for the message. The count is saved and also stored as a temporary counter. The address pointer is initialized to the start of the input buffer and if the byte count is non-zero, the ioinp flag is set.

Once the flag is set, subsequent interrupts cause the input bytes to be stored in the input buffer until all the bytes have been input. When the entire message has been received, the ioinp flag is cleared and the input is sent to the spint() task via the spispi queue.

4.3.6 Service-Processor Output Task and Interrupt Handler

The service-processor output task, spoutt(), receives data from three queues - spispo, audspo, and graspo. It merges these data into a double-buffered output array and calls the interrupt handler to start the transmission. A maximum of 255 bytes can be transferred at one time to the service processor; if the combined input from the queues is more than 255 bytes, spoutt() will make more than one call to the interrupt handler, until all the queues have been emptied. The logic of spoutt() has been set up to take one message at a time from all the queues instead of emptying one queue before going to the next. This was done so that an audio message would not get backed up behind a long string of graphic messages.

4.3.6.1 The Service-Processor Output Task (spoutt())

After waiting for the initialization task, spoutt() initializes the counts of its double buffers, resets the pointer to the buffers, and clears the I/O flag. It initializes an array of pointers to the input queues and gets the value of the auxiliary control port, which has been set in the INIT task. This is done because the interrupt handler must set and clear one bit of this port without changing the other bits. After its initialization is complete, spoutt() waits until the first message from the service processor is received before it sends anything.

After the first message has been received from the service processor, spoutt() enters an infinite loop in which it checks its input queues and if nothing has been input, it suspends itself. If there is an entry, it enters a do-while loop which continues until all the input has been processed. The input is processed by extracting one message at a time from each queue which contains an entry. The messages are added to one of the output buffers until either the output buffer is filled or all the messages have been extracted. This output buffer is sent to the interrupt handler by the call to spout(b) and then the buffer pointer is switched to the other buffer. The double-buffering technique allows one buffer to be transmitted while the other is being filled in preparation for transmission.

There is one condition that requires special handling by spoutt(). It is the selective transmission of messages based on the setting of the Bendix front panel mode switch. The mode switch has three 'on' settings. They are weather radar only, combination weather radar/AID, and AID only. In general, regardless of switch setting, tasks within the user processor function as if audio and video data blocks are always to be sent to the service processor. In actuality, the data blocks are always sent as far as spoutt(). Then spoutt(), with the help of the teu() task, determines which audio and video data blocks should be sent on to the service processor. Audio data blocks in response to keystrokes are always sent. Target-related audio and video data blocks are sent always in AID only mode, never in weather radar only mode, and sometimes in combination mode.

In more detail the process is as follows. Each time there is a change in the front panel switch setting, the service processor sends this setting to the user processor. It ultimately is passed to the teu() task where it is used by the modeck() function. Modeck() is called once per scan before any audio or video data blocks are sent to spoutt() for this scan. Modeck() sends a 'user mode switch' data block to spoutt(). The user mode switch has one of two settings: AID only or weather radar only. If the front panel setting is AID only, the user mode switch setting will be AID only. If the front panel setting is weather radar only, the user mode switch setting will be weather radar only. If the front panel setting is combination mode, the user mode switch setting will be weather radar only except when one of the following conditions is met: (1) there is a threat or prethreat to be displayed, (2) a threat or prethreat has been displayed within the last 8 seconds, or (3) the EXT key has been pressed on the keyboard. If one of these conditions is met, the user mode switch setting is AID only.

When spoutt() recognizes a user mode switch message in its input data stream, it uses this to set its internal flag "uswitch". Thereafter, whenever there is an entry in one of spoutt()'s input queues, spoutt() uses "uswitch" to decide whether to flush the entry or to send it on to the interrupt handler for transmission to the service processor. If "uswitch" = weather radar only, the entry will be flushed. If "uswitch" = AID only, the entry will be sent on. Regardless of "uswitch" setting, three message types are always sent to the service processor. They are the scale message, slave acknowlege message, and user mode switch message.

4.3.6.2 The Transmission Startup Routine (spout(b))

Spout(b) is used by the spoutt() task to start the transmission of a message to the service processor. It is called with the pointer (index) of the buffer to be output and has the facility to retransmit a message if a transmission timeout occurs. This was included because the service processor is interrupted only on the first byte of the message and if that interrupt is missed for some reason, the message must be retransmitted from the start. Thus, if spout(b) is called by spoutt() and I/O is still in progress, spout(b) waits until it is awakened either by a timeout or by the completion of the I/O. If the timeout occurred, the previous message is retransmitted.

If the last message was transmitted properly, any pending timeout messages are cleared and then the byte count of the current message is checked. If the byte count is not legal, spout(b) merely returns, thus ignoring the message. If the byte count is okay, it then prepares to set up the transmission. Interrupts are disabled at this point because the alternate register set, used by the interrupt handler, is set up during this time. A spurious interrupt would cause the set up to be erroneously done.

The alternate registers are used for the output interrupt handler so that this interrupt can be handled as fast as possible. The hardware of the user processor is set up so that the service processor is put into a wait state each time it tries to input a byte from the user processor and remains in the wait state until the interrupt routine in the user processor outputs the byte. The use of the alternate registers allows the interrupt routine to save and restore the state of the machine as fast as possible in the Z80.

Register C is set to the output port address, register B is set to the byte count, and HL is set to the address of the byte to be output. This setup allows the interrupt routine to use the OUTI instruction. Next the byte count is output to the transmission port and then bit 7 of the auxiliary control port is set, which causes the service processor to be interrupted. (See Sierra documentation.)

After the transmission has started, spout(b) requests a timeout message from the timer task and if not a transmission, returns to the spoutt() task.

4.3.6.3 The Service Processor Output Interrupt Handler (spoh())

The interrupt routine switches to the alternate register set, which has been initialized by the startup routine, spout(b). Register A is output to the auxilary control port to make sure that the service processor is interrupted only on the first byte. (Register A had been left by spout(b) to contain the proper value to be output.) Next the Z flag is tested to determine if the transmission is complete. (Note that the OUTI instruction will set the Z flag when the byte count goes to zero.) If there is more to do, it outputs the next byte using OUTI and returns. If the transmission is done, it clears the I/O flag and wakes the output task, spoutt().

4.3.7 The Timer Task (stim()) and Interrupt Handler (ctc())

Stim() and ctc() together provide interval timing for tasks within the user processor. They also maintain a one-second timer, "sectimr", which is a global parameter that may be referenced by other tasks.

The slave single-board computer contains a chip that supplies four counter-timers. It is initialized in init() to produce an interrupt every 62.5 milliseconds. When this interrupt occurs, control is passed to the interrupt handler, ctc(), which simply places a one-byte dummy message into the timtim queue and wakes the timer task, stim().

In addition to being awakened by ctc() at regular intervals, stim() can be awakened by a task wishing to initiate or halt a timer. Stim() maintains an array tcount[], each element of which serves as a timer for one of the user-processor tasks. A task initiates a timer by loading a delay count into its timer output queue and waking the timer task. The delay count is from 1 to 255, with each count = 62.5 mseconds. Stim(), when awakened, places this delay count into the task's tcount[] entry. Thereafter, each time stim() is awakened by ctc(), it will decrement each active tcount[] entry. When a count reaches zero, the corresponding task is sent a message in its timer input buffer, the task is awakened, and the timer is deactivated (set to -1). A task may stop its timer at any time by sending stim() a -1 count.

Stim() first checks three queues for input: initim, teutim, and spotim, from tasks init(), teu(), and spoutt(), respectively. If an input is present, it is used to update the tcount[] array. Next stim() checks for an entry in the timtim queue, signalling a 62.5-msec wake-up from ctc(). If an entry is present, all active tcount[] entries are decremented. Tasks whose timers time-out are sent a one-byte message and awakened. Finally, stim() increments "sectimr" every 16th ctc() wake-up, to keep elapsed system time since power-up, lsb = 1 second.

5.0 THE AUDIO RECORDING AND AUDIO RAM LOADING FUNCTIONS

A program has been written to facilitate the recording of spoken words and phrases and generated tones. Another program was written to store the generated data into audio RAM banks in the AID system during its initial program load sequence.

The audio recording program is formed by linking the following relocatable files using indirect command file LZBLD.CMD:

```
AUDBLD.R
AUDCOM.R
AUDREC.R
AUDBITS.R
CT.Z
MT.Z
CHDR.Z
```

The first four are application files; CT.Z and MT.Z are C-compiler libraries which provide console and disk I/O interfaces; CHDR.Z provides the call/return interface between the CP/M operating system and the initial C-function, main().

The audio recording program provides a menu-driven interface for the operator. It interfaces to the Continuously Variable Slope Delta Modulation (CVSDM) audio recording and playback S-100 boards. It is also capable of processing previously recorded data contained in a floppy disk input file and of storing old or newly recorded data in an output disk file. Input file entries and newly recorded data may be annunciated, edited, bypassed or output to the output file. Thus, the entries in a previously recorded audio file may be deleted, edited or passed on to the output file. In addition, newly recorded entries may be inserted between previously recorded entries in the output file. A provision is also present for quickly bypassing N records of the input file. Finally, a capability is present for synthesizing audio tones of operator-specified frequency and duration.

The input and output data files contain variable-length records. Each record consists of a fourteen-byte header and a variable-length (4095 bytes max) data array. The record format is specified using the C "union" and "struct" data structures as follows:

```
union{
     struct{
             int entpres          ;
             int lngth            ;
             char audnam[10]      ;
             char auddat[4096]    ;
     }audent                      ;
     char filrec[1]               ;
}un = 0                           ;
```

Parameter "entpres" is used simply to verify that a valid entry is present. "Lngth" specifies the length of the audio data stored in array "auddat[]". Array "audnam[]" contains an operator-selectable name for the recorded audio. Array "filrec[]" overlays the "audent" data structure and provides a means for easily moving records.

The linked output from LZBLD.CMD is downloaded to the Z80 software development facility and converted to a COM file. It is then renamed to AUDBLD.COM. The program is run by operator command:

```
A> AUDBLD F1.T F2.T
```

where F1.T is the input audio data file and F2.T is the output file (any file names may be substituted). Both files must be specified. If no input data is available, a dummy F1.T file should be specified. A "null" file, F3.T, is present on the floppy disk. It may be used to clear a data file, as follows:

```
A> PIP F1.T = F3.T
```

Note that a program calling arguments can't easily be run under the ZSID debugger. When debugging, the conditional compile flag "FIXED" in file AUDBLD.C should be set. The program will then automatically use F1.T for input and F2.T for output without specifying them as arguments in the call.

5.1 The Audio Build (AUDBLD.C) File

This file contains the main() function, which is the first function run in any C program (called by CHDR.Z).

The main() program opens the input and output files, presents the operator with prompts for selecting program operations and then closes I/O files when the program terminates.

The first operation performed is to open the I/O files. Depending on the setting of the conditional compile switch "FIXED", these files may be operator specified in the program initiation statement or the pre-specified files F1.T and F2.T. In the former case a test is made to make sure the operator specified the correct number of files (two).

A set of prompt statements are then output to the operator informing him of the options available. They include: "R", record new data, "I", input next record from input file, "B", bypass N records in input file and "Q", quit — return to CP/M. The program proceeds, based upon the operator response. After each operation has been performed, the operator is again presented with the options menu.

If the operator chose to record new data, the record() function is called. If (s)he chose to input a record, the input() function is called. In this case the function returns a one if a record is present. The operator is then prompted to determine if the record should be sent to the output file or ignored. If it should be output, the output() function is called. If input() returned a zero then no more entries exist in the input file. The operator is so informed.

If the operator chose to bypass input records then (s)he is prompted to specify the number to bypass. That number of records are then input (by calling function input()) automatically. Note that function input() displays the name attached to each record as it is read. If less than the specified number of records exist in the input file, the operator is so informed.

The operator may also choose to quit, in which case the I/O files are closed and control is returned to CP/M. Finally, if the operator enters an illegal response (s)he is so informed.

5.2 The Audio Communication (AUDCOM.C) File

This file contains the functions that input/output audio data records from/to the disk and communicate with the console and annunciator card.

5.2.1 The Disk Input (input()) Function

This function inputs an audio record from the input disk file and sends its identifying character string to the console. Upon operator direction it then sends the recorded audio to the annunciator card where it is annunciated. The function returns a one if a record was found, a zero if none was present (the last record had been read) and a two if the record could not be read.

The first operation performed is to read the record header from the disk. The length of the record's data area may then be used to load the data. If either of these reads fails, the operator is so informed and the function returns a two. Next, the function sends the record's name and size to the console. The operator is then prompted to see if the data should be sent to the annunciator. Note that this operation is bypassed when the operator has chosen to bypass N records. The function returns a one when a successful record read has been accomplished.

5.2.2 The Audio Annunciation (annun(audptr, audlng)) Function

This function sends audio data to the annunciator card and activates the annunciator. The data to be sent starts at address "audptr"; "audlng" bytes are sent.

The first operation performed is to reset the annunciator card. This is accomplished by outputting a byte (any byte) to port OX4F. The "audlng" bytes, starting at "audptr" are then output using the otir() function to perform the actual transfers. This function uses the fast block move Z80 instruction "OTIR" for this purpose. Port OX4D is used. Since otir() may move a maximum of 256 bytes at a time, multiple calls may be needed. Finally, the annunciator is started by outputting a byte (any byte) to port OX4E. The function then returns.

5.2.3 The Disk Output (output()) Function

This function outputs the record currently stored in union "un" to the output disk file.

The first operation performed is to inform the operator of the size of the record to be output. The record is then output. If the output operation fails the operator is so informed and is asked if another attempt should be made.

5.2.4 The Operator Prompt (prompt(msgptr,retflg)) Function

This function outputs the ASCII string pointed to by "msgptr" to the console. If "retflg" is one, it waits for the operator to press a key. If the key entered is a 'Y' the function returns a one. If the entered key is an 'N' it returns a zero. Entering any other key causes an error prompt to be sent to the operator. If "retflg" is zero the function simply returns after outputting the prompt.

The first operation performed is to send the message pointed to by "msgptr" to the console. Note that C library function lenstr() is used to determine the length of the string.

If a response was requested ("retflg" non zero) then a character is input from the keyboard. Note that the C library function getch() is used and that a loop is needed to "fix" it. The loop removes any left over line feed (OXOA) characters from the input buffer.

The character received is tested to see if it was a 'Y' or an 'N'. If it was, a one or a zero, respectively, is returned. Otherwise, an error message is sent to the operator. The function will not return until a legal key is pressed. However, the error prompt is output only once.

5.3 The Audio Recording (AUDREC.C) File

This file contains the functions required to record and edit audio data. It also contains the function that generates a tone record.

5.3.1 The Audio Record (record()) Function

This function supervises the recording of data received from the CVSDM card, the editing of the data, its playback by the annunciator and finally, its output to the output disk file.

The CVSD card always records 4095 bytes of data (about 1.6 seconds of speech). Z80 assembly language function getaud() inputs this data and stores it in array crsddat[]. The data in this array may then be edited (i.e., starting and ending bytes may be specified) by manipulating pointers "strptr" and "endptr". When the operator is satisfied with the results, the edited data may then be copied into the file record buffer un.audnet and output to the output disk file.

The first operation performed is to compute starting (stradd) and ending (endadd) address for the cvsddat[] array. Then the operator is sent a series of prompts specifying the options available. These include: "S", start recording, "T", generate a tone, "E", edit, "P", playback, "O", output a record and "Q", quit.

When the operator presses "S", the getand() function is called and audio data is recorded and placed in cvsddat[]. Flag "recflg" is set to indicate that audio has been recorded, the start and end pointers, "strptr" and "endptr" are initialized to "stradd" and "endadd", respectively, and the function redisplays its options prompts.

If the operator presses "P", the recorded data is played back through the annunciator. The "recflg" is checked first to make sure recorded data is present. If it is not, the operator is so informed. The function's option prompts are then redisplayed.

If the operator presses "E", the "recflg" flag is checked. If it is zero it means that no data recorded using the "S" option exists in un.audent. However, it may contain a record received from the disk input file. This data will then be edited and the operator is so informed. Start and end pointers "strptr" and "endptr" are set and the data is read from un.audent to cvsddat[]. Flag "recflg" is then set, since "recorded" data now exists in cvsddat[] and the edit function edit() is called. When edit() returns, the record() function's option prompts are redisplayed.

If the "recflg" flag was set when "E" was entered, then edit() is called directly.

The operator may also enter "T" to record a tone. In this case "recflg" is set and the tone() function is called. When it returns and the options prompts are redisplayed. Note that a tone record may also be edited.

Once a new record has been satisfactorily created it may be output to the output disk file by pressing "O". This operation tests "recflg" to make sure a new record exists. If one doesn't the operator is so informed. If one does, the audout() function is called to output the record. The options prompts are then redisplayed.

Finally, when the operator has completed recording operations (s)he may return to the main menu by pressing "Q".

5.3.2 The Audio Output (audout()) Function

This function builds an audio record in un.audent from data in array cvsddat[] and the audio nametag supplied by the operator. It then outputs the record to the output disk file.

The first operation performed is to get the audio nametag from the operator and store it in the record header. The tag can be up to nine characters in length. Next the "lngth" and "entpres" fields in the header are set. Then the audio data in array cvsddat[] is moved to the record's auddat[] array. Finally, the output() function is called to output the record to the output disk file.

5.3.3 The Audio Editing (edit()) Function

This function displays the current start and end indexes for the audio data in array cvsddat[]. It then requests index changes from the operator. Finally, it checks to see that the end index is greater than the start index. If not, it re-requests index values.

The first operation performed is to compute cvsddat[] array indexes "stridx", "endidx" from pointers "strptr", "endptr" and cvsddat[] start pointer, "staadd". It then displays these indexes on the console.

The operator is then prompted to input, if desired, a new start index and end index. These values are then tested to be sure they do not exceed 4094. A test is then made to insure that the end index is not less than the start index. Finally, new start and end pointers to the cvsddat[] array are computed and the new data size is displayed on the console. The function then returns.

5.3.4 The Tone Generator (tone()) Function

This function generates data bytes in edit file cvsddat[] representing a continuous tone, as specified by the operator. The operator is prompted for tone frequency and duration. The data space after the tone is filled with zeros (silence).

The first operation performed is to prompt the operator to input the tone frequency. If the value specified is out of range (20 to 2000 Hz) the operator is so informed. The operator is then prompted to enter tone duration in tenths of seconds. Again, if the value input is out of range (0 to 16) the operator is informed and prompted to re-enter the value.

The program then computes the number of 19.7 kHz (the sampling rate) samples in a half cycle of the tone and the number of samples in the tone's duration. Note: Tone duration was limited to 1.6 seconds so that the total number of samples would fit in a 16-bit, signed parameter "timidx" (16x1970 = 31520), thus avoiding the use of double-precision arithmetic. A tone duration of 1.6 seconds is long enough for all applications. Some loop control flags are then set before proceeding to the tone sample generation loop.

Before the tone sample loop is started the operator is informed that calculations are proceeding. This "human factors" prompt was found to be necessary, since the loop takes a noticeable time to complete. The loop is then entered.

"timidx" tone samples are generated and stored in bytes in array cvsddat[]. An inner loop counts through the number of samples in each half cycle of a tone. If amplitude flag "amp" is one, then ones are stored in the bytes in cvsddat[]; otherwise the bits are left zero. Flag "amp" is toggled after each half cycle of samples has been generated. When timidx is not divisable by eight, a partially loaded byte remains. That byte is then output to cvsddat[].

The start and end pointers "strptr" and "endptr" are then set so that the edit() function can be used to edit the tone data, if desired. Finally, the remaining bytes in the 4K cvsddat[] array are zeroed. This produces a "silent" period after the tone which may be "edited into" the record if needed. The function then returns.

5.4 <u>The Sample Bits Recording (AUDBITS.Z80) File</u>

This function inputs audio sample bits from the CVSD audio recording card, packs them into bytes and stores the bytes in array cvsddat[]. This function also provides the sampling clock signal at a 19.7 kHz rate that is needed by the CVSD card. This timing is dependent upon the execution time of the major loop within the program, itself. In fact, certain statements are included simply to adjust sample timing. The program is written in assembly language and assembled using the AZ80.CMD assembler indirect command file.

The first operation performed is to read the start address for cvsddat[], "staddd", into the DE register pair. Eight data bits will be shifted into a byte within the sampling loop. A bit count of eight is placed in the C register to control this loading operation. Next, the sampling loop is entered and a CVSD card sample pulse-high is output. The audio data sample is then input and shifted into the current byte (pointed to by registers DE) in array cvsddat[]. The byte count in C is then decremented. If the current byte has been completely filled, the bit counter is set back to eight and a sample clock pulse-low is output. A check is then made to see if the end of the cvsddat[] array has been reached (check DE against "enddat"); if it has, the program returns. If it hasn't, the program delays a fixed number of clock cycles and returns to the start of the sampling loop.

If the current byte has not yet been filled, some path delay instructions are executed, a sample clock pulse-low is output, some more delay statements are executed and the program returns to the beginning of the sampling loop.

Note that the timing in this program has been "fine tuned" to produce the required 19.7 kHz sampling frequency. Program changes could alter this timing and cause the audio recording system to malfunction.

5.5 <u>The Audio RAM Loading (AUDRAM.C) File</u>

This file contains the functions required to load the previously recorded audio data into the audio RAM banks. The AUDRAM.COM function is called as part of the AID's initial program load sequence. If more than one 16K audio RAM bank exists, then separate versions of AUDRAM.COM are called. Each version is tailored to load a particular bank with its proper audio data.

5.5.1 The RAM Loading (AUDRAM.COM) Function

This function operates in two modes. If it is called with a file specified as an argument, e.g.:

```
A>AUDRAM F2.T
```

then it will load the audio data bytes from the records in the specified file into an internal 16K-byte array and then return to the CP/M operating system. The operator may then save the new version of the program (with the 16K array loaded) by entering:

```
A > SAVE 111 AUDRAM.COM.
```

This "loaded" version is then ready to be called as part of the AID's initial program load sequence.

In the second mode, which runs when AUDRAM.COM is called without a file specified in the call, it transfers the audio data bytes from the 16K internal array to the specified audio RAM bank.

The first operation performed is to check to see if an input file was specified in the program call line. An additional check is made to make sure that only one file was specified.

If a file was specified, it is opened. The contents of the file are read (minus the record headers) into array annunc[]. A check is made during this process to make sure the array's 16K size is not exceeded. If it is, the operator is so informed, the input file is closed and the function returns. If the array size is not exceeded, the function exists normally by closing the input file and returning.

If no input file is specified, the function moves the contents of annunc[] to the specified RAM bank. This is accomplished in machine code using the block move instruction, LDIR. Note that if the destination RAM bank is the upper 16K of the master's RAM, then the move is straightforward with destination starting address 0XC000. However, if the audio RAM bank is one of the 16K RAM banks on the 64K RAM board, then the on-board 16K bank must be deselected and the off-board bank selected before the block move is performed. The on-board bank is then reselected.

5.5.2 The Audio Record Input (input()) Function

The AUDRAM.COM function must read records from the specified input file. It does this by calling the input() function.

The first operation performed is to read the record's entry-present flag and record-length field (the first four bytes). If this read fails, the operator is so informed. It then checks to see if the file is empty or if no records remain (entry-present flag not 'A'). If this is true, the function returns a zero.

If the file is not empty the remainder of the record is read. If this read operation fails, the operator is so informed and the function returns a 2. If the record read was successful, the function returns a 1.

APPENDIX A

AID Operating System

1.0 INTRODUCTION

The purpose of this appendix is to present a description of the system executive used in the AID master and slave single board computers. The system executive to be described consists of two major components: a task scheduler and a set of queue access functions. The scheduler initiates application tasks on a priority basis in response to wakeups from interrupt handlers and tasks. Messages are passed between tasks and between interrupt handlers and tasks by means of circular queues. The queue access functions provide standardized access to these queues. The task scheduler will be described first, followed in later sections by a description of queue data structures and queue access functions.

2.0 SCHEDULER

Task scheduler designs may be grouped into two general catagories: pre-emptive and nonpre-emptive. A task running under a pre-emptive scheduler may be suspended if a task of higher priority is awakened by an interrupt handler. This is important for systems in which data received by an interrupt handler must be processed immediately by a task awakened by that handler. Pre-emptive scheduling may also be necessary if the data rate is such that there is the possibility of data being overwritten before it can be processed. This problem can sometimes be solved by double buffering the data either in hardware or in software.

By contrast, a task running under a nonpre-emptive scheduler may not be pre-empted (that is, suspended) even though a task of higher priority is awakened. Except for interrupt servicing, the running task has control of the computer until it suspends itself. In systems using non pre-emptive scheduling it is necessary for tasks to cooperate in using processor bandwidth by limiting the amount of processing performed between voluntary suspensions. This is not a serious limitation in many real-time applications in which timing requirements are not too critical. A significant advantage is that it considerably simplifies the scheduler design and therefore reduces memory and execution time requirements.

Another major advantage of nonpre-emptive scheduling is that it reduces the possibility of inconsistent data being passed between program components. Presumably, a task will complete the output (or input) of an entire data message before voluntarily suspending. By contrast, a task running under a pre-emptive scheduler may have processed part of the data in a message when an interrupt handler causes a higher priority task to run. This task might change the content of the message that was being processed by the original task. A data access lockout mechanism must be implemented to avoid this problem[1].

[1]The use of queues to pass all data between program components also reduces the possibility of this type of error, since new data does not overwrite old data until the space has been released.

An objective of this design was to keep the scheduler as simple as possible so that executive execution time overhead would be minimized. In addition, the anticipated applications did not require the immediate processing of data received from interrupt handlers. As a result, a nonpre-emptive type scheduler was chosen.

2.1 Task Control

The status of each task is maintained in a Task Control Block (TCB). The TCBs for all application tasks are contained in a linked list data structure, as shown in Fig. A-1. A TCB contains a forward link pointer, used by the scheduler to access TCBs, the starting address of the task, the task's current stack pointer and "status" and "signal" flags.

A linked list data structure is used for the TCB data area primarily for the purpose of determining task priority and to facilitate scheduler operations. When a task suspends itself the scheduler always starts checking TCBs at the beginning of the linked list. As a result, the task described by this TCB has highest priority. Scheduler operations are also facilitated by linking the lowest priority TCB to the highest. Then, during periods when no tasks are scheduled, the scheduler simply searches continuously through the TCBs. Note that this will be the normal idle condition for the task scheduler unless a lowest priority idle task is defined. The idle task must be designed such that it pauses periodically to allow higher priority tasks to run.

The linked list data structure for TCBs is also convenient for systems in which tasks install other tasks to run or in situations in which task priorities must be changed dynamically. The forward link pointers may simply be changed to reorder the list.

A task may be in one of three states: running, waiting or ready, as shown in Fig. A-2. Its current state is determined by the values of its "status" and "signal" flags. A task's state may be changed by calling one of the three functions: run(), sleep() or wake(). Note that the "signal" flag for a RUNNING task may be in one of two states. This means that a RUNNING task may also be in the READY state. This situation can occur when a RUNNING task is interrupted and scheduled to run again. More will be said about this later.

The scheduler and each task maintain their own stack areas. When a task is interrupted or suspended, its context (that is, the processor's registers and flags) is stored on its stack. Similarly, when the scheduler transfers control to a task, the scheduler's context is stored on its stack. The context is restored when control is returned to the task or scheduler. When a high level language is used, the scheduler may use the stack area originally allocated by the compiler; task stacks must be explicitly declared as data areas in the program.

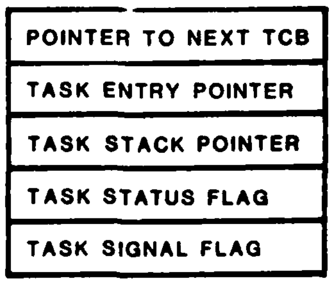


Fig. A-1. Task control block.

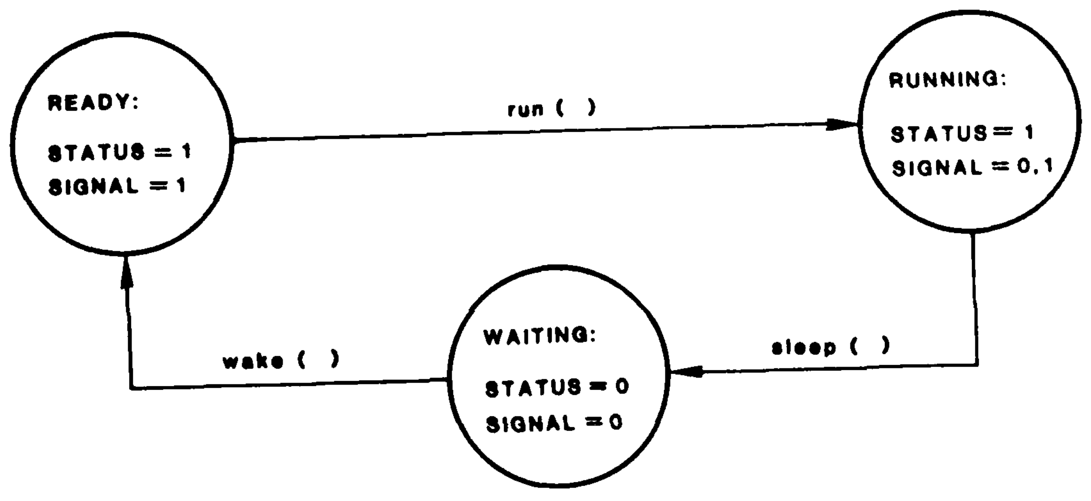


Fig. A-2. Task states.

When a task is invoked, the scheduler's stack pointer is saved in a memory location. The task's stack pointer is then read from its TCB and loaded into the Z80's SP register. When a task is suspended the reverse operation is performed. On the other hand, when an interrupt occurs, the context of the running program (task or scheduler) is saved on its stack, but the interrupt handler uses the running program's stack for its operations. It must, of course, POP off all data that it pushes onto this stack before returning to the interrupted program. Interrupts are disabled while the handler is saving and restoring the interrupted program's context. However, they may be enabled while it is performing other operations, since subsequent interrupt handlers will merely stack the context of the handler they interrupt.

A task program has the general structure shown in Fig. A-3. It is a C function containing an "infinite" loop. During program startup each task is run from its beginning to the point where it first suspends itself (i.e., calls sleep()). During this time the task may perform any task-specific initial setup operations. This would include the initialization of any data items that do not have to be reinitialized during a restart (restart is performed by an initialization task which will be described later). After startup, task entry and exit operations are performed entirely within sleep().

2.2 Scheduler Functional Components

The scheduler is comprised of four basic functions: sched(), wake(), run() and sleep(). Their relationships to task states are shown in Fig. A-4. Each is a C function; some contain machine code.

After initialization, the scheduler scans TCBs until it finds one with a "status" flag set. It then clears the corresponding "signal" flag and calls run(). Run() transfers control to the previously suspended task and resumes task operation at the point where suspension occurred.

A task suspends itself by calling sleep(). This function clears the task's "status" flag and transfers control back to the scheduler. However, it first checks the "signal" flag. If it is set it means that the RUNNING task was also READY. That is, while the task was running, an interrupt occurred. The interrupt handler rescheduled the running task to run again. The "signal" flag indicates this condition. The sleep() function handles this situation by resetting the task's state to READY (by setting the "status" flag) and returning control to the scheduler. The scheduler then scans TCBs, starting at the highest priority, until it finds a READY task and invokes it.[2]

[2] Note that the order in which the operations are performed in sleep() is important. For example, it would appear that STATUS could be reset after the SIGNAL test, when the SIGNAL test fails. However, if an interrupt occurs between the test and the reset, then STATUS and SIGNAL will both be set (if the interrupt handler reschedules the running task). Then, when STATUS is reset after returning from the interrupt, the final state will be: STATUS = 0, SIGNAL = 1, which is a disallowed state. Programming sleep() as shown will avoid this problem.

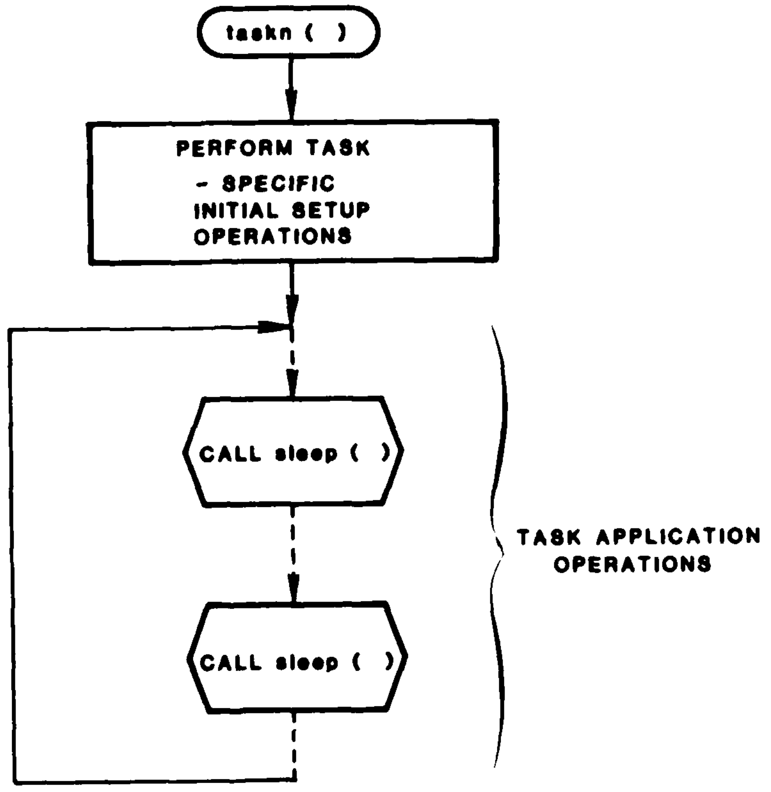

taskn ( )
PERFORM TASK - SPECIFIC INITIAL SETUP OPERATIONS
CALL sleep ( )
CALL sleep ( )
TASK APPLICATION OPERATIONS

Fig. A-3. Task program structure.

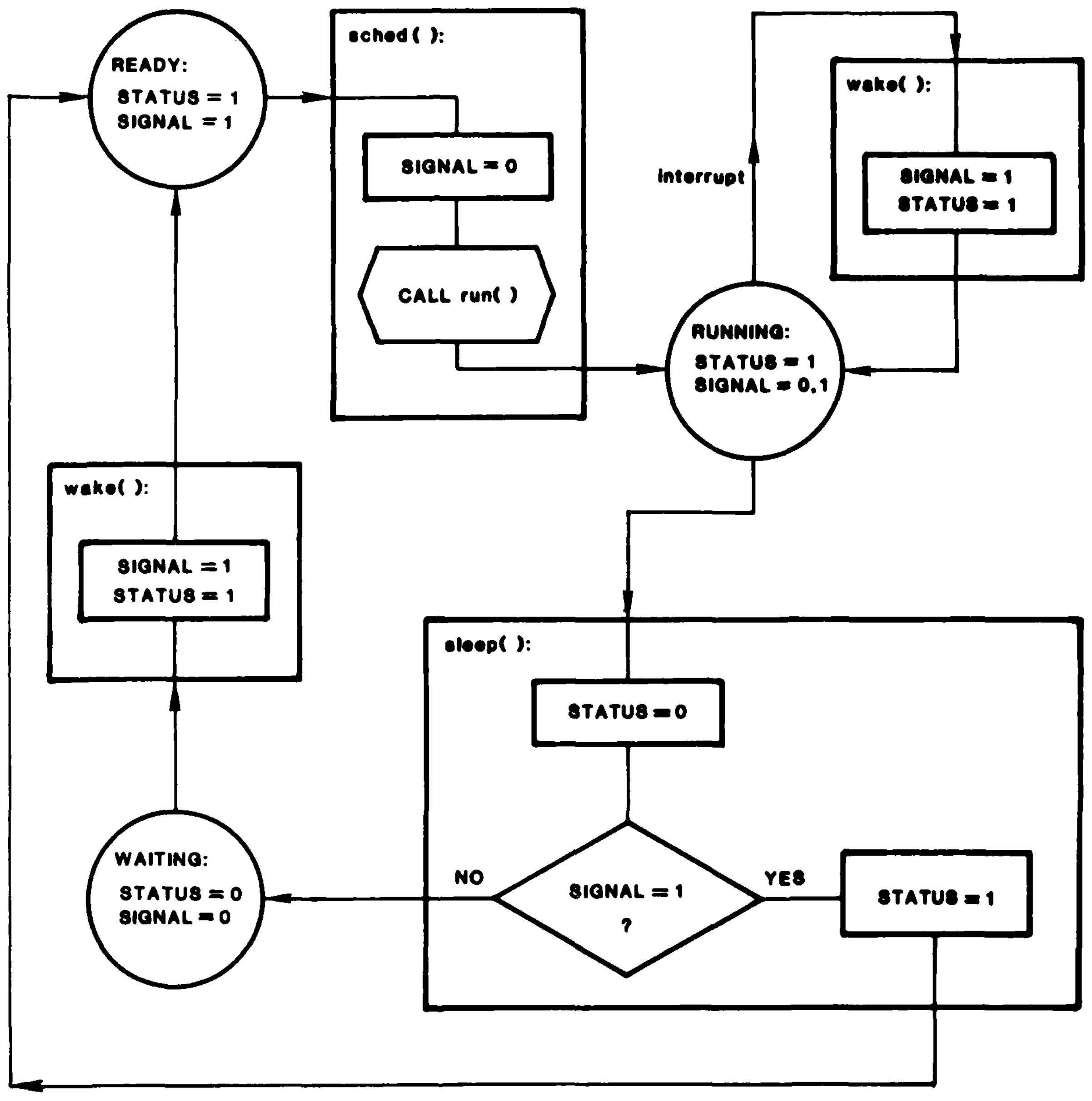


Fig. A-4. Task states and state change mechanisms.

A task may be awakened from the WAITING state by calling wake(). This function's single argument specifies the number of the task to be awakened. It is used as an index into the array of TCBs. Wake() sets the task's "status" and "signal" flags. These are the only operations performed by wake; it is written entirely in C. Wake() may be called either from an interrupt handler or from a task.

One additional scheduler program component is the pause() function. It was not included as a "basic" function since it can be derived from the wake() and sleep() functions. It provides a means for a task to voluntarily give processor control back to the scheduler but, before doing so, reschedule itself to run again. This gives higher priority tasks that may have been awakened by interrupt handlers a chance to run before the pausing task resumes. The pause() function may be implemented simply as a C function that calls wake() and then sleep(). The task number needed as an argument in the call to wake() may be supplied either as an argument in the pause() call or, since the number of the currently running task is known to the scheduler, it may be supplied automatically.

2.3 Scheduler Initialization

All C programs start at the beginning of the function called main(). In this application main() performs initial startup operations that do not have to be performed during a program-controlled restart.

One operation performed by main() is to initialize run() and sleep(), as shown in Fig. A-5, by computing addresses RUNADR and SLPADR, respectively. These addresses are needed when control is passed between run() and sleep() during invocation and suspension of tasks, as will be explained later. Since these addresses will be referenced from outside their respective functions, they must be computed and treated as global data items.

As mentioned earlier, each task must be called and run to the point at which it first suspends itself. This operation is diagramed in Fig. A-6. In main(), return address STRADR is computed and stored in variable "rtnadr". It is used as a return address by sleep() during task initialization operations. Then main() saves its stack pointer and gets the stack pointer for the first application task (in the figure, taskn() represents "task n") from its TCB.

It then calls the task using a normal C function call operation. The task initializes startup data, as described earlier, and suspends by calling sleep(). Function sleep() saves the task's stack pointer in its TCB and passes control back to main() via the address in "rtnadr". Main() then restores its stack pointer and repeats the entire sequence of operations for each application task. At this point sleep()'s return address, "rtnadr", is switched to RUNADR, the entry point in run(). During all future operations sleep() will return to this point.

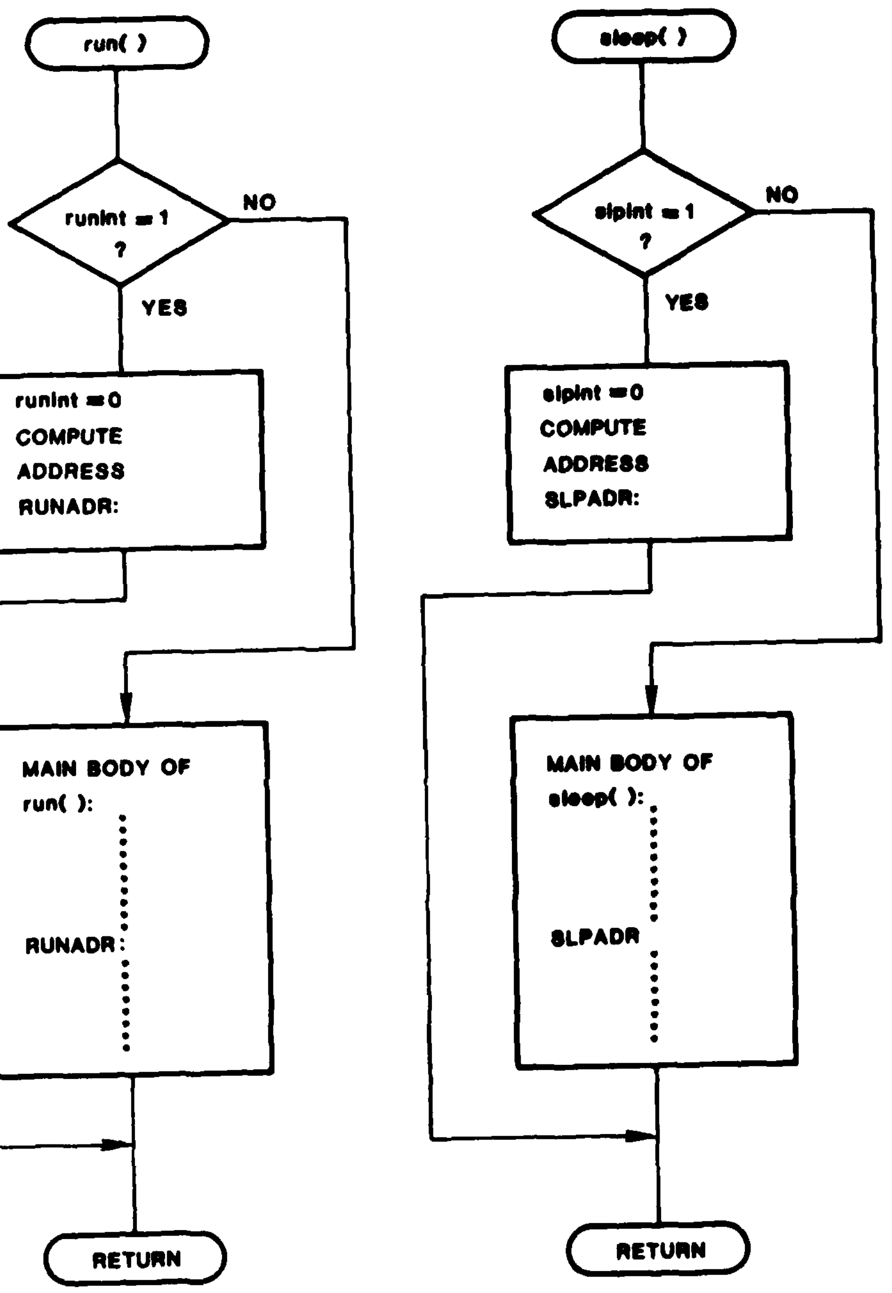


Fig. A-5. Computing return addresses: RUNADR, SLPADR.

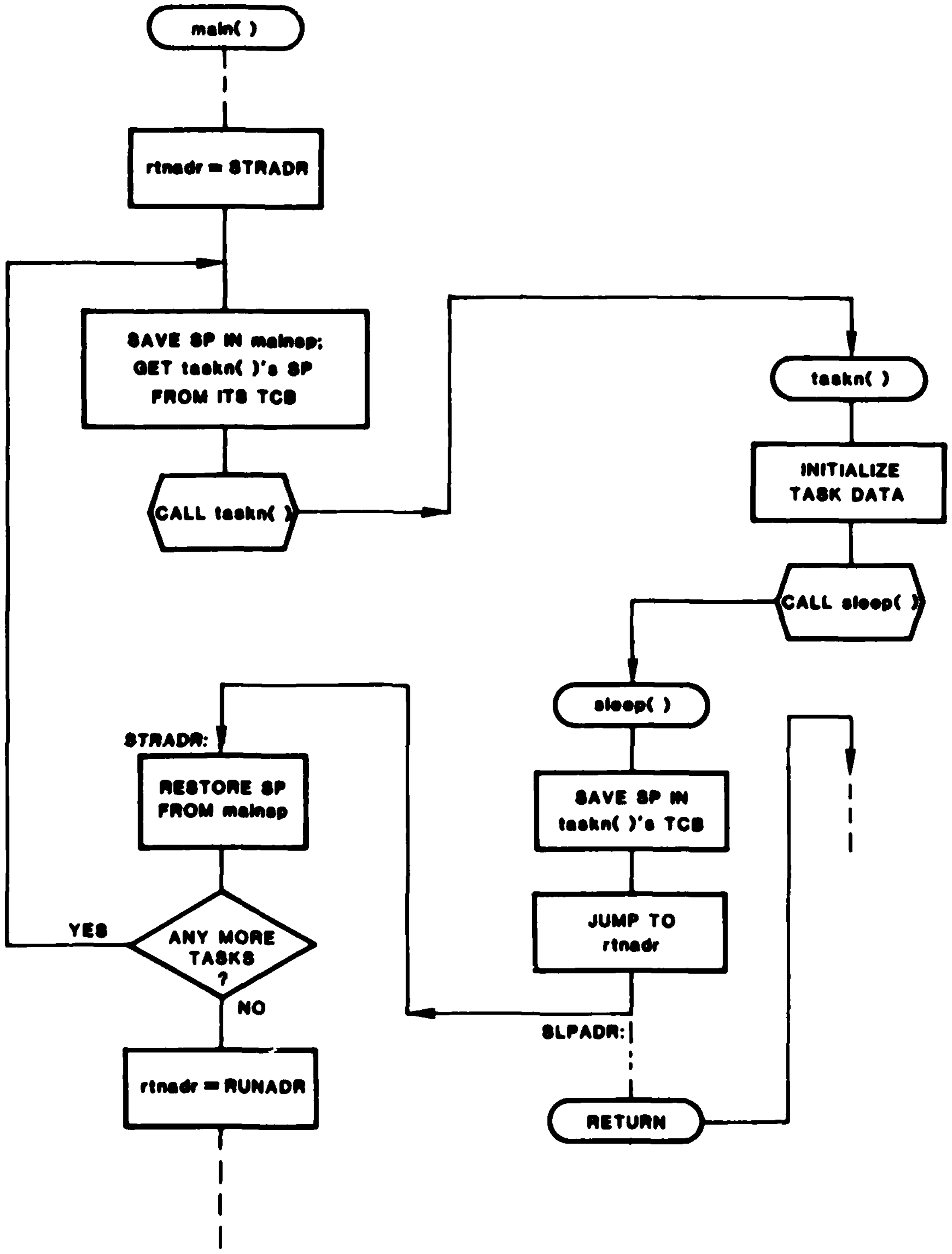


Fig. A-6. Task initialization.

2.4 Scheduler Operation

After initialization, scheduler operation proceeds as diagramed in Fig. A-7. Function sched() scans TCBs, starting at the beginning of the linked list, until one is found in which the "status" flag is set. It then stores the index of the selected TCB in variable "tcbidx" and calls run(). Run() saves the scheduler's context by switching to the second set of registers provided by the Z80 (it also pushes registers IX and IY on its stack).[3] It then saves the current stack pointer and transfers control to address SLPADR in sleep(). Sleep() gets the stack pointer from the TCB pointed to by "tcbidx" and places it in the Z80's SP register. It then restores the task's context from its stack and returns to the task by means of the normal C function return protocol. This is possible since the return address was pushed on the stack when the task was initially run to its first call to sleep(), as described earlier.

At this point the task proceeds to perform application operations until it again calls sleep(). As shown in Fig. A-7, after sleep() clears the "status" flag it checks the "signal" flag. If it is set it proceeds, as described earlier, to reschedule the task. In either case it then saves the task's context on its stack, stores the task's stack pointer in its TCB and transfers control to address RUNADR in run(). run() then restores the scheduler's stack pointer and its context and returns to the scheduler via the normal C return protocol. The scheduler then starts at the top of the TCB linked list and scans for another task to run.

Note that as far as the task is concerned, its call to sleep() and the return were just the same as any other C function call. *It is not "aware" of* the fact that sleep() transferred control back to the scheduler and that other tasks possibly ran before sleep() returned. The same is true of sched() and its call to run(). The scheduler is not "aware" that run() transfers control to a task and receives control back before returning. It is therefore unnecessary for the person writing an application task to know the operational details of the task scheduler; (s)he simply programs a call to sleep() to cause a suspension and expects the task to be awakened at the next C instruction.

Programming interrupt handlers is somewhat more complicated. First, the context of the interrupted function must be saved on the currently active stack. This is performed using PUSH instructions in machine code. After performing I/O operations the context is POPed off the stack and control is returned to the interrupted function.

[3] If the second register bank is needed for some other purpose, the scheduler's entire context may be saved on its stack. For example, the second register bank could be used to save context in interrupt handlers and thereby minimize their execution times.

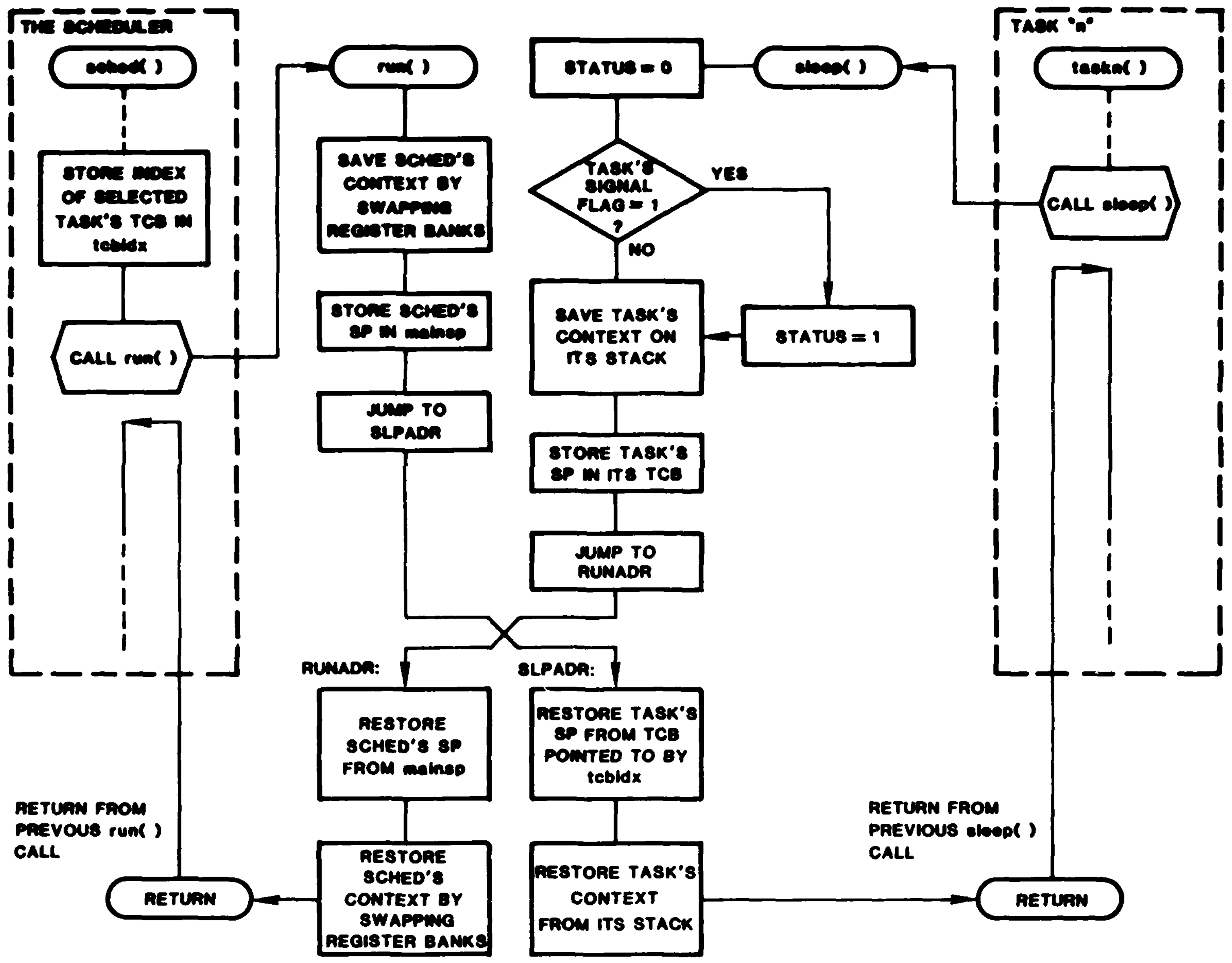

THE SCHEDULER
sched( )
STORE INDEX OF SELECTED TASK'S TCB IN tcbidx
CALL run( )
RETURN FROM PREVOUS run( ) CALL
RETURN
run( )
SAVE SCHED'S CONTEXT BY SWAPPING REGISTER BANKS
STORE SCHED'S SP IN mainsp
JUMP TO SLPADR
RUNADR:
RESTORE SCHED'S SP FROM mainsp
RESTORE SCHED'S CONTEXT BY SWAPPING REGISTER BANKS
STATUS = 0
TASK'S SIGNAL FLAG = 1 ?
YES
NO
STATUS = 1
SAVE TASK'S CONTEXT ON ITS STACK
STORE TASK'S SP IN ITS TCB
JUMP TO RUNADR
SLPADR:
RESTORE TASK'S SP FROM TCB POINTED TO BY tcbidx
RESTORE TASK'S CONTEXT FROM ITS STACK
sleep( )
TASK "n"
taskn( )
CALL sleep( )
RETURN FROM PREVIOUS sleep( ) CALL
RETURN

Fig. A-7. Task scheduling and return.

The interrupt handler is written as a normal C function (with in-line machine code). However, it is not entered or exited using the normal C protocol. Instead, the interrupt vector is computed such that the first instruction performed is the first PUSH of the context-save sequence. This bypasses the normal C function entry sequence. It is necessary to save the context before any other operations occur so that the interrupted program's context will not be lost. Similarly, control is transferred back to the interrupted program with a normal Z80 return-from-interrupt (RETI) instruction just after the context has been POPed from the stack, bypassing the normal C function exit sequence.

An interesting observation can now be made concerning this particular implementation of a task scheduler. It has not been necessary for the implementor to be aware of the details of the C function entry or exit protocols used by the compiler. In interrupt handlers, they are simply bypassed. The running program's context (machine state) is saved, interrupt operations are performed, the context is restored and control is returned to the running task before the handler's normal C function return operations. Similarly, transfers between the scheduler and tasks are performed within C functions run() and sleep(). These functions save machine context on the currently active stack, transfer control and restore the context of the destination function from its stack. In this way task scheduling and interrupt operations are transparent to the compiler. The C program's context is saved before these operations are performed and restored afterward.

3.0 QUEUES

The other aspect of real-time program design involves the implementation of a means for passing data between tasks and between interrupt handlers and tasks. Since tasks and interrupt handlers run asynchronously, the use of queues for these operations insures that messages will not be lost. A discussion of the subject of queues, queue access functions, and their interaction with the task scheduler will be presented in this section.

3.1 Queue Structures

Within C it is possible to define a queue header structure data type by using a "typedef" declaration. For a variable-entry-size queue, this might take the form:

```
typedef struct{
    int    head         ;
    int    tail         ;
    int    length       ;
    char   task         ;
    char   *pbuf        ;
}QUE
```

This declaration may be placed in an "include" file that is attached to each program source file. Entries are added to the queue starting at the "tail" pointer and removed starting at the "head". The queue's length is specified by parameter "length". The parameter "task" will be described in more detail later when queue access functions are described. Briefly, it contains the number of a task that suspended itself when it was not able to complete a requested queue access operation. Pointer "pbuf" points to the beginning of the actual queue data array. The first byte in each queue entry contains the number of bytes that follow in that entry.

Using the QUE data type it is then possible to define queues and their headers as follows:

```
#define TKLNGTH = 6
QUE timkey = {0,0,TKLNGTH,0}  ;
char tkbuf[TKLNGTH]           ;.
```

This might define, for example, a queue for passing data between a timer task and a keyboard task. The pointer "pbuf" to the queue array tkbuf[] must be initialized to the address of the array with a statement of the form:

```
timkey.pbuf = tkbuf           ;.
```

This queue may be referenced from the file containing the timer or keyboard task by first declaring the queue header as an external reference:

```
extern QUE timkey             ;
```

and then by referring to its address, &timkey, in, for example, the argument list of a queue access function.

3.2 <u>Queue Access Functions</u>

Various queue access functions may be written to satisfy different application requirements. These functions may be built upon two primitive functions which will be called putq() and getq().

The putq() function declaration has the form:

```
putq(source, dest, count)
```

where "source" is a pointer to an array (or item) of data to be placed in the queue and "dest" is a pointer to the queue header. "Count" is the length of the message to be moved, in bytes. For example, this function could be used to move a four-byte message from array gmttim[] in the timer task to the keyboard task by writing:

```
if (putq(gmttim, &timkey, 4) == -1)
        sleep()               ;     .
```

As indicated, putq() is programmed to return a -1 value if not enough room exists in the specified queue to store a message of length four (including one more byte for the entry size). In the case shown the programmer has simply chosen to suspend the task if this situation arises.

The function getq() is declared similarly:

```
getq(source, dest).
```

It also returns a -1 if no entry is present in the queue pointed to by argument "source".

Using these two primitive functions to perform the actual queue access operations it is possible to write other useful functions that have general application within a task scheduling environment and support the orderly flow of messages between tasks. For example, a function may be designed such that when an attempt is made to enter a message into a full queue, the task will be suspended. Later, when a message is removed, the suspended task will be awakened so that it can store its message. The suspended task identifies itself by storing its task number in the queue's header in item "task". Such a function might be declared as follows:

```
putqwt(source, dest, stask, count).
```

The function name suggests that the task will put a message in the queue if room exists but will wait (suspend, call sleep()) if not enough room exists. Parameter "stask" specifies the number of the calling task[4]. Similarly, a function to remove messages from a queue, getqwt(), may be designed such that when a task attempts to remove a message from an empty queue it will suspend. When a message is later placed in the queue the suspended task will be awakened.

By using putqwt() and getqwt() it is possible to control the execution of tasks in response to the availability of messages to process in their input queues and the availability of space to store messages in their output queues. For example, if an output device becomes momentarily blocked, the queues that feed it messages will become "backed up" and will cause the corresponding tasks to suspend. When the device becomes unblocked the tasks waiting to send data will be awakened in an orderly manner, based upon the availability of storage space in their respective output queues. The flow of messages is similar to the flow of automobiles in a traffic tie-up on a major highway.

4.0 SYSTEM OPERATION

The previous sections have described the design and operation of the task scheduler and queue management functions. What remains is to describe the sequence of operations that occur during the startup of an application program and to briefly discuss some typical application tasks.

[4]Note that the number of the currently running task is available in global parameter "tcbidx" so that it need not be specified as an argument. This would eliminate a possible source of programming error.

After the initialization of each task, as described earlier, main() calls wake() to schedule the initialization task, init(), and sets the TCB linked list pointer to zero so that the scheduler will test init()'s TCB first (init() is the highest priority task). Main() then calls sched() and, from this point on, control is never passed back to main() unless the system is reset from hardware. Sched() then checks init()'s TCB and passes control to it.

Task init() performs all initialization operations that must be performed first during startup and later during restart. That is, init() is programmed so that if another task or interrupt handler wakes it, it will restart the application program from the beginning. It performs these operations with interrupts disabled and it enables interrupts just before it suspends itself.

The first operation performed in init() is to link all the TCB's together in a loop. This is accomplished by setting the forward link pointers in the TCBs. As described earlier, the last (lowest priority) TCB is linked to the first so that the scheduler will loop looking for READY tasks. Init() then performs other application-specific restart initialization operations and finally suspends itself by calling sleep(). These operations may involve waking application tasks. However, if other tasks are not awakened the scheduler will simply loop until an interrupt handler wakes a task.

Tasks and interrupt handlers provide the various data processing and control services required by the particular application. For example, most real-time applications will be required to service interrupts from a hardware timer device. The interrupt handler wakes a timer task which, in turn, provides interval timing services to other application tasks. Applications that provide a man-machine interface will usually need to service a keyboard. The interrupt handler for this device places the received key-stroke character in an output queue and wakes a keyboard task. This task will then input the character from the queue and process it. Tasks and interrupt handlers may also be defined to process data from other sources such as communication channels or measurement sensors.

Tasks processing data inputs will usually wake tasks that provide data to output devices. These might be video or alpha-numeric displays, communication channels or equipment controllers. The typical output device is designed to receive a byte of data and then produce an interrupt when it is ready to receive the next byte. The interrupt handler must therefore be designed to output subsequent bytes until the entire message has been sent. It then wakes the output task. The output task must initiate the transfer by sending the first byte and then suspend and wait to be awakened by the interrupt handler. It may be assured that the wakeup was from the interrupt handler if the handler sends a "signal" byte message to the task before waking it. The task, on being awakened, checks the queue from the interrupt handler and possibly other input queues to determine the source of the wakeup.

APPENDIX B

"C" To Z80 Assembly Optimization

1.0 INTRODUCTION

The purpose of this appendix is to provide a brief guideline for generating and optimizing the Z80 assembly language output of Vandata's "C" compiler and translator. The reader is assumed to be familiar with the C programming language and with Z80 assembly language. In addition, the reader is assumed to be familiar with the use of Vandata's compiler under the RSX-11M operating system.

1.1 Procedure

The C source to be optimized should be compiled as usual and thoroughly tested before the optimization procedure is started. When this has been done, the first step in optimizing is to generate the Z80 assembly language from the C source code. This is accomplished using the indirect command file CZ80A.CMD, which goes through all the phases of the compiler to the point where the asharp code has been generated and then invokes the translator to convert asharp to Z80 assembly. The output assembly language file has the file name extension .ASM.

The next step is to use PIP to copy the .ASM file to a file with .Z80 as an extension. All subsequent modifications are done to the .Z80 file. This step is done for several reasons: first, it leaves the original assembly language source intact for comparisons to the edited version; second, it provides a way to know if the assembly source has been optimized; and third, it provides some safety in preventing the deletion of the hand-optimized file if, for example, the CZ80A.CMD command file is invoked again.

The .Z80 file is then edited and when the optimizations required are accomplished, the AZ80.CMD command file is invoked to assemble the file and generate the relocatable object file. (If a listing file is desired in addition, use AZ80W.CMD.) Again, the modified object should be thoroughly tested. When this is done, the .ASM file may be deleted since it can be regenerated easily from the C source and is no longer useful.

1.2 Optimization Guidelines

The primary reason in the AID application for optimizing the C code is to increase the speed of execution. Appropriate optimization can usually gain a factor of two in execution speed but much consideration must be given to what parts should be optimized and how in order to obtain the benefits. Efforts in improving the original C algorithm will often result in higher performance than hand optimizing a poor algorithm.

1.2.1 Tuning the C Source

In addition to the proper choice of algorithm, there are a number of considerations at the C source level that will aid in the subsequent optimization of the Z80. First, the C source should be divided into relatively small and simple modules in order to make the optimization editing more tractable and easier to follow. A two-page C source file may be eight or more pages long when translated to assembly language and because of the nature of the compilation process, the generated assembly code will not necessarily follow the C source line for line, especially with deeply imbedded logical structures. On the other hand, it may not help much to optimize procedures which are too small because the overhead in calling and returning from such routines could account for a large part of the time spent in them.

Secondly, one must give some consideration to the variables in the routine. As far as possible all variables should be declared as characters (1 byte) and local variables should be declared static. Static variables are easier to locate and eliminate the relatively long indexed stack instructions, but using static local variables will render a subroutine nonrecursive.

Thirdly, care should be used in selecting arithmetic operations. For example, multiplication always calls the library multiplication routine, whereas left shifts are encoded in-line with add instructions. Thus, if multiplication by a power of two is needed, use the left shift operator instead.

There are a number of other ways to help "tune" to C source for later optimization. For example, the ordering of statements affects the way in which the compiler generates its output. Such considerations, however, are too specific to discuss here, and can be learned through experience.

1.2.2 Optimizing the Z80 Assembly Language

Once the assembly code has been generated, care must be given to what sections should be modified. Obvious candidates for optimization are loops and code inside loops, arithmetic expressions, array indexing expressions, complex logical tests, and procedure calls. The following discussions are intended as general considerations and cannot cover all the possible means of optimization.

1.2.2.1 Arithmetic Expressions

One of the specifications of C is that in arithmetic expressions, all 8-bit quantities are converted to 16-bit quantities and 16-bit arithmetic is performed, even if all the quantities are 8 bits. On an 8-bit machine like the Z80, the overhead involved in doing 16-bit arithmetic is considerable, and can be eliminated if it is not necessary. Here is a typical example.

```
C Source              Z-80 OUTPUT             OPTIMIZED Z-80

char na, nb ;         LD    H,20H             LD  A,(na.)
nb = 32 -na ;         LD    A,(na.)           LD  B,A
                      LD    E,A               LD  A, 20H
                      ADD   A                 SUB B
                      SBC   A                 LD  (nb.), A
                      LD    D,A               10  sec (@ 4 MHz)
                      LD    A,L
                      SUB   E
                      LD    L,A
                      LD    A,H
                      SBC   D
                      LD    H,A
                      LD    A,L
                      LD    (nb.), A
                      20  sec (@ 4 MHz)
```

Multiplication and division by powers of two should be done by adding and shifting instead of the calls to the C library functions. Here is a typical example:

```
C Source              Z-80 Output             Optimized Z80

setups (x,y,nb,nl)    LD    L,(IX+0AH)        LD A,(IX+0AH)
int x,y ;             LD    H,(IX+0BH)        SRL A
char nb,nl ;          PUSH HL                 LD (h.),A
                      LD    HL,02H            10 μsec
static char n ;       PUSH HL
n = nl/2 ;            CALL c.idiv
                      POP HL
                      LD    A,L
                      LD    (n.),A
                      30 + μsec (not including time in c.idiv)
```

(Note that even though nl was declared a character argument, C always passes argument values as 16-bit integers).

There are many other ways of improving arithmetic expression coding, and some experience is needed to be able to understand why the compiler sometimes generates very obscure code and to be able to optimize the coding appropriately. Needless to say, a primary consideration in modifying such code is that the result should be the same as in the unmodified version. Here is another example where the compiler must perform unnecessary operations to perform 16-bit logic where only 8-bit is required. It also shows how register usage may be improved.

```
C Source                  Z80 OUTPUT                OPTIMIZED

                          CALL c.ent0               CALL c.ento
#define MASK0 0x70        LD   A, (ccolor.)         LD A, (ccolor.)
                          CP   0f0H                 CP 0F0H
#define MASK1 0xF0        JR   Z,.35                JR z, .35
                          LD   A, (ccolor.)         LD (.74), A
#define WHITE 0xF0        LD   (.74),A              AND 70H
                          LD   A, (ccolor.)         CPL
extern char ccolor ;      LD   C,A                  AND 0F0H
                          ADD  A                    LD L,A
                          SBC  A                    LD H,0
                          LD   B,A                  CALL colorg
stbyt()                   LD   L,C              .35 JP c.ret0
                          LD   H,B                  17  sec (not including
                          LD   A,L                            calls)
                          AND  70H
static char scolor ;      LD   L,A
                          LD   A,H
if (ccolor! = WHITE)      AND  00
                          LD   H,A
                          LD   A,L
                          CPL
scolor = ccolor ;         LD   L,A
                          LD   A,H
colorg ((~(ccolor &       CPL
 MASK0)) & MASK1);        LD   H,A
                          LD   A,L
                          AND  0F0H
                          LD   L,A
                          LD   A,H
                          AND  00
                          LD   H,A
                          CALL colorg
                     .35: JP c.ret0
                          44 μsec (not including calls)
```

1.2.2.2 Loops

Loops are an obvious area for optimization. If it is known that a loop index will always be less than 128, it should be declared as a character rather than an integer. Even when declared as a character some time can be saved in incrementing and checking the loop index as the following example demonstrates:

```
C Source                  Z80 OUTPUT                 OPTIMIZED Z80

static char i, nb ;       SUB   A                    SUB A
                          LD (.34),A                 LD (.34),A
for (i=0; i<nb; i++) .55:LD HL, nb.            .55:  LD HL, nb.
{body of loop}            LD A,(.34)                 CP (HL)
                          CP (HL)                    JP P,.75
                          JP P,.75                   (body of loop)
                          (body of loop)             LD A,(.34)
                          LD A,(.34)                 INC A
                          ADD 01H                    LD (.34),A
                          LD (.34),A                 JP .55
                          JP .55               .75:
                      .75:
```

While in this example only 4 μsec per iteration is saved, it shows one type of optimization that can be performed. If the above loop was executed 100 times inside of a loop that executed 1000 times, a total 4/10 second could be saved. In a real-time program, such a savings could be crucial.

1.2.2.3 Logical and Arithmetic Tests

Normally, the code generated for tests is quite compact, but it is not unusual to encounter jumps to jumps, unnecessary register manipulations, and other time wasting instructions. Here is an example where an integer on the stack is compared to -1.

```
      Z80 OUTPUT             Improved              Optimized

      LD A,(IX+04)           LD A,(IX+04)          LD A,(IX+04)
      CP OFFH                CP OFFH               INC A
      JR NZ, .2              JR N2, .11            JR NZ, .11
      LD A,(IX+05)           LD A,(IX+05)          LD A,(IX+05)
      CP OFFH                CP OFFH               INC A
.2:   JR NZ, .11             JR NZ, .11            JR NZ, .11
       (body)                 (body)                (body)
.11:  (continue)      .11:   continue       11:     continue
```

The above "optimized" version could be further improved if the integer was expected to be -1 more often than not. It is left to the interested reader to find the improvement.

1.2.2.4 Other Areas of Optimization

There are a number of other areas where code optimization can be applied. For example, the compiler does not do an optimum job at allocating and using registers. Many such cases are obvious, but in complex expressions, it may require considerable thought and effort to improve the code. How much effort should be applied to extract as much as possible from code optimization must be answered from the overall programming effort.

1.2.3 Warnings, Bugs, and Disclaimers

There is one known bug in the Vandata Z80 assembler which must be mentioned - it does not flag some relative addresses properly to the linker. In particular, this means that one should not use the CASE statement in the C source of routines to be optimized because the CASE statement generates a table of addresses. When this table goes through the Z80 assembler and the linker, only the address offsets get generated in the actual code, which causes the program to jump to the wrong place.

APPENDIX C

Aural Alerting for Phase I AID System

The AID alerting system is based primarily upon the guidelines for TCAS alerting developed in simulation at the Boeing Commercial Airplane Company. Sounds employed in the alerting system are stored in digital form in 48K of RAM. Both voice and non-voice sounds are employed. The user-processor single board computer contains logic which determines when to annunciate each aural alert message.

Aural alerting phrases are listed in Table C-1. Definitions of the siren, C-chord and chime are provided in Fig. C-1. These sounds are identical to those used in the Boeing simulation.

Figures C-2 and C-3 define the voice alerting messages which correspond to each of the possible IVSI commands. In the event of both up-sense and down-sense limit rate advisories, the aural messages for each will be concatenated with a short pause.

Figure C-4 provides a flowchart of the logic used for resolution advisory alert processing. This logic is called once per scan.

Figure C-5 provides a flowchart of the logic used for controlling alerts associated with traffic advisories. This logic is entered once per scan per target. It is entered after resolution advisory alert processing has been completed.

Figure C-6 provides a high-level flowchart of the logic used to determine whether or not a particular target received from the CAS logic should be selected for possible display.

Figure C-7 provides a high-level flowchart of the logic used to determine if a selected target can be displayed on-screen or, if it is off-screen, whether to use an off-screen symbol or to simply delete the target.

Some principal design characteristics of this alerting system are:

1. The siren is annunciated once at the beginning of a sequence of RA indications on the IVSI.

2. After the siren, a voice message which corresponds to the types of RA's present is repeated continuously until manually cancelled by the crew.

3. After cancellation, the voice alert message (unaccompanied by the siren) will sound once each time the state of the IVSI changes. Only the RA-sense which has changed is annunciated.

4. A C-chord is sounded when a target transitions to prethreat (amber) status from a lower priority status. However, this alert is suppressed if an uncancelled aural alert for an RA is being annunciated.

TABLE C-1.

Aural Alerting Phrases Available in Phase I AID System

(Note: Some of these phrases are not employed in the Phase I AID system, but have been provided to facilitate future modifications.)

VOICE	NON-VOICE
ABORT	Beep
ALERT	Buzz
ALTITUDE	C-chord
CAUTION	Chime
CLEAR	Chirp
CLIMB	Pause
DESCEND	Siren
DESCENT	
DON'T	
FEET	
FIVE	
HUNDRED	
LIMIT	
MAINTAIN	
ONE	
PER MINUTE	
TCAS	
TEST	
THOUSAND	
TRAFFIC	
TWO	
WARNING	

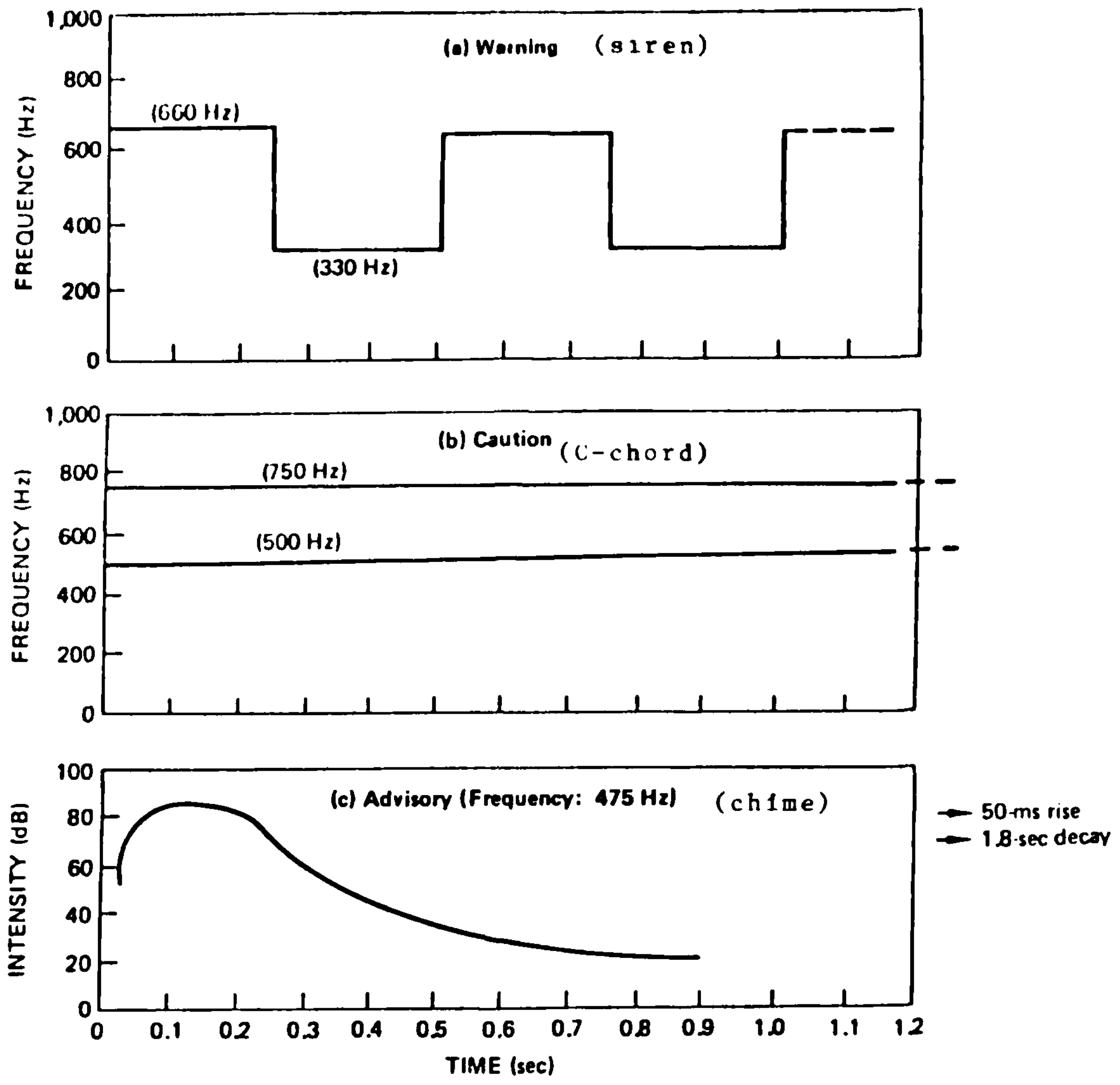


Fig. C-1. Alerting sounds.

RA	VOICE MESSAGE	CAS/IVSI LAMPS									
		10	9	8	7	6	5	4	3	2	1
1	CLIMB	1	0	0	0	0	0	0	0	0	0
2	DESCEND	0	1	0	0	0	0	0	0	0	0
3	DON'T CLIMB	0	0	1	1	1	1	0	0	0	0
4	LIMIT CLIMB	0	0	1	1	1	0	0	0	0	0
5	LIMIT CLIMB	0	0	1	1	0	0	0	0	0	0
6	LIMIT CLIMB	0	0	1	0	0	0	0	0	0	0
7	DON'T DESCEND	0	0	0	0	0	0	1	1	1	1
8	LIMIT DESCENT	0	0	0	0	0	0	0	1	1	1
9	LIMIT DESCENT	0	0	0	0	0	0	0	0	1	1
10	LIMIT DESCENT	0	0	0	0	0	0	0	0	0	1
11	MAINTAIN CLIMB	0	0	0	0	0	1	1	1	1	1
12	MAINTAIN CLIMB	0	0	0	0	1	1	1	1	1	1
13	MAINTAIN CLIMB	0	0	0	1	1	1	1	1	1	1
14	MAINTAIN DESCENT	0	0	1	1	1	1	1	0	0	0
15	MAINTAIN DESCENT	0	0	1	1	1	1	1	1	0	0
16	MAINTAIN DESCENT	0	0	1	1	1	1	1	1	1	0
17	TCAS ABORT	1	1	1	1	1	1	1	1	1	1

Fig. C-2. Voice Messages For Resolution Advisory Alerts.

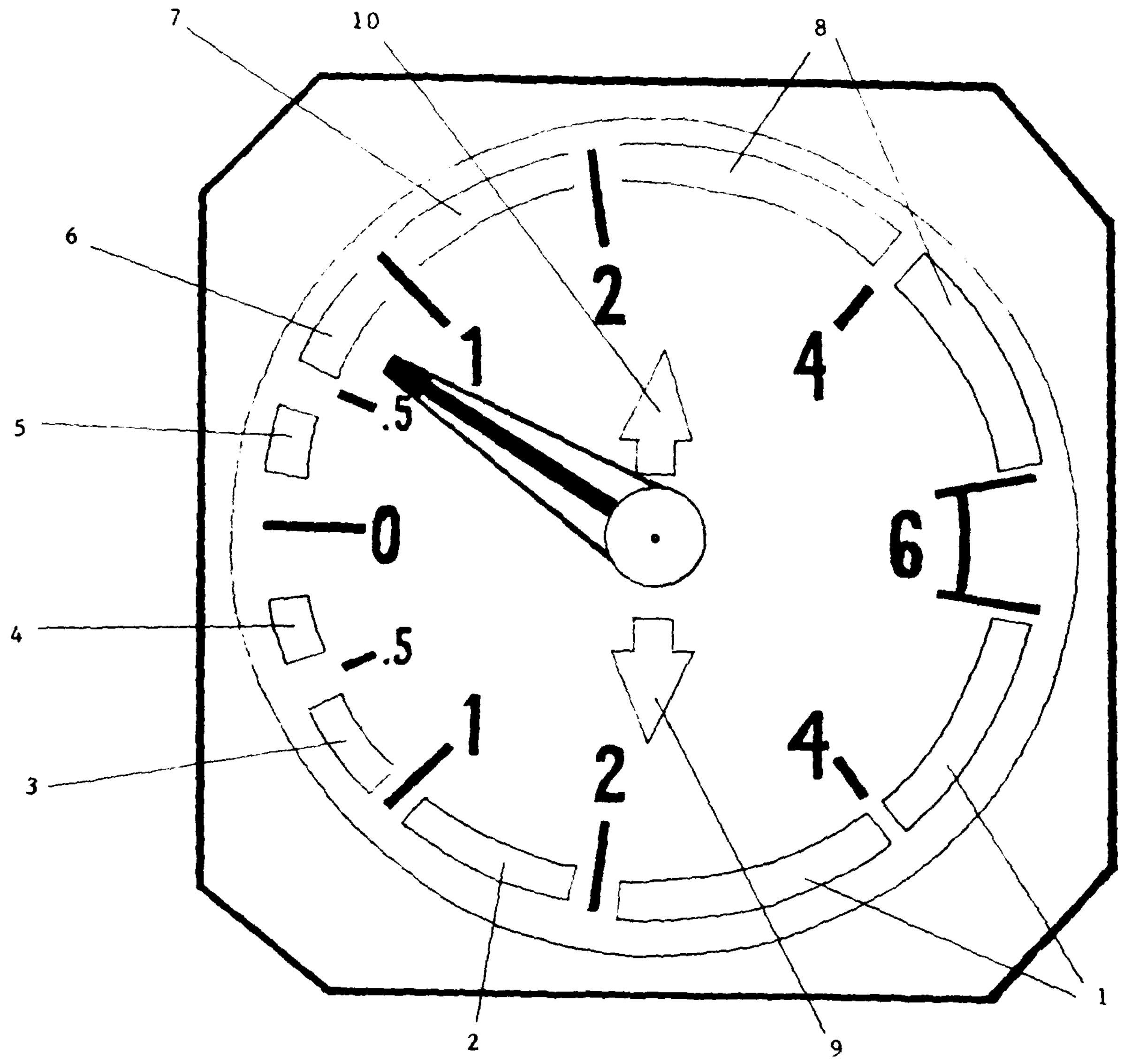
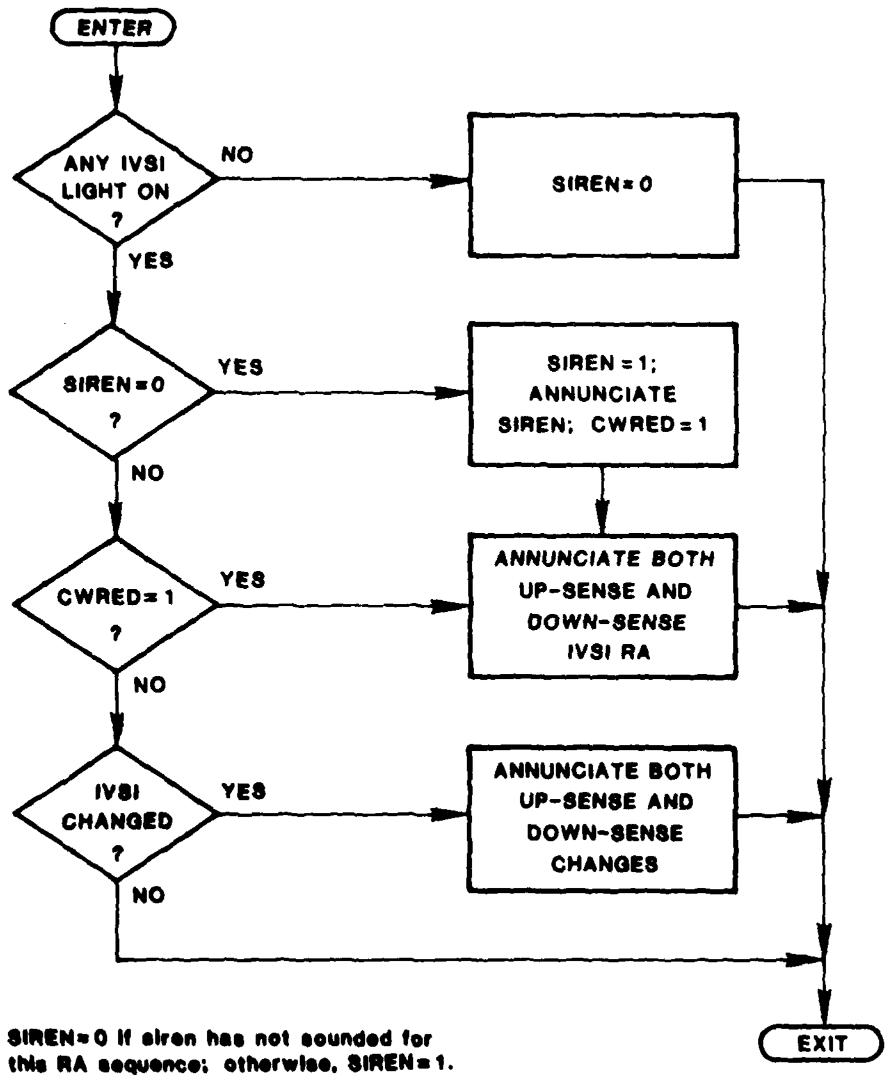


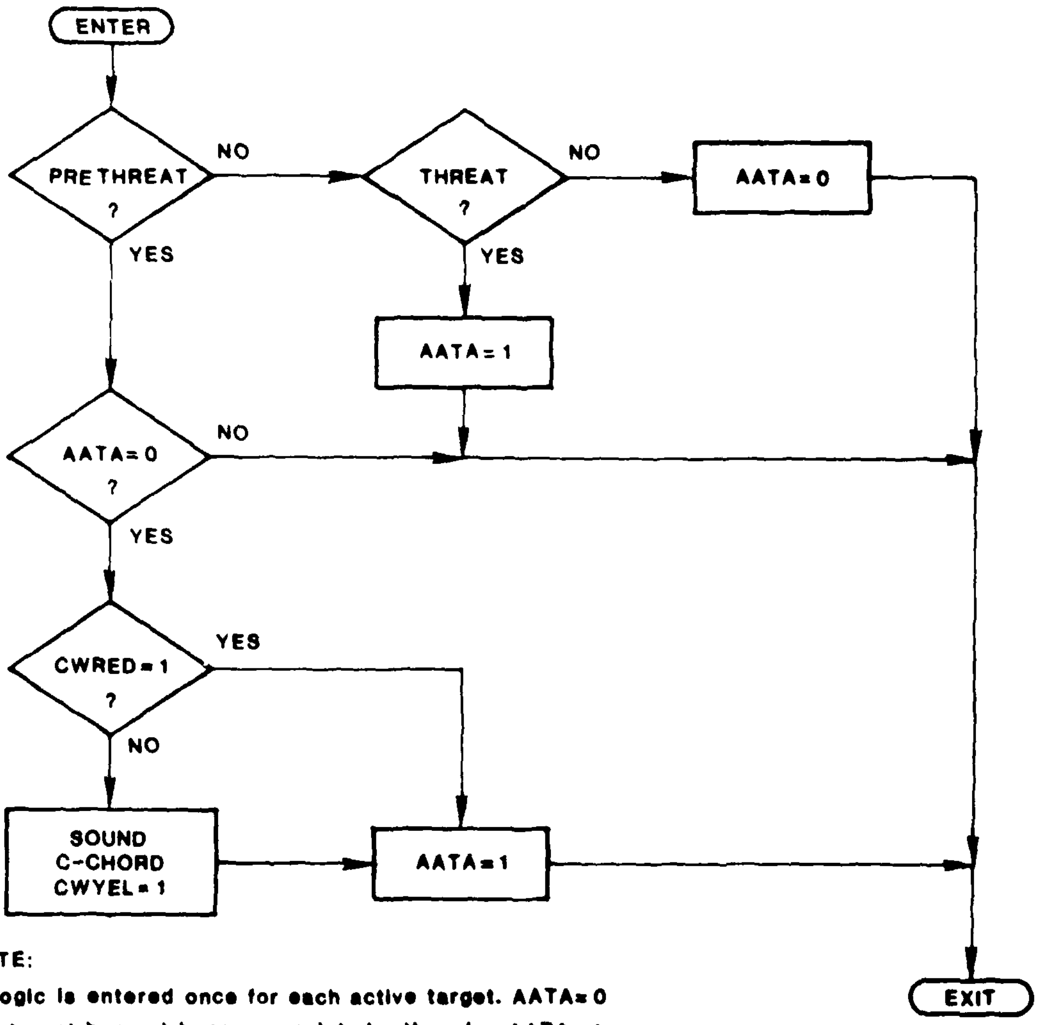
Fig. C-3. IVSI lights numbering scheme.


ENTER
ANY IVSI LIGHT ON ?
NO
YES
SIREN=0
SIREN=0 ?
YES
NO
SIREN=1; ANNUNCIATE SIREN; CWRED=1
CWRED=1 ?
YES
NO
ANNUNCIATE BOTH UP-SENSE AND DOWN-SENSE IVSI RA
IVSI CHANGED ?
YES
NO
ANNUNCIATE BOTH UP-SENSE AND DOWN-SENSE CHANGES
EXIT

SIREN=0 if siren has not sounded for this RA sequence; otherwise, SIREN=1.

CWRED=0 if red C/W light is off; otherwise, CWRED=1. Light can be cancelled manually or by software.

Fig. C-4. Resolution Advisory alert processing.



NOTE:

Logic is entered once for each active target. AATA=0 if target has not been annunciated; otherwise AATA=1. Threat aural alerts serve to annunciate all threats and pre-threats on display at the time the threat appears. CWYEL=1 if amber Caution/Warning light is to be lit.

Fig. C-5. Traffic advisory alert processing.

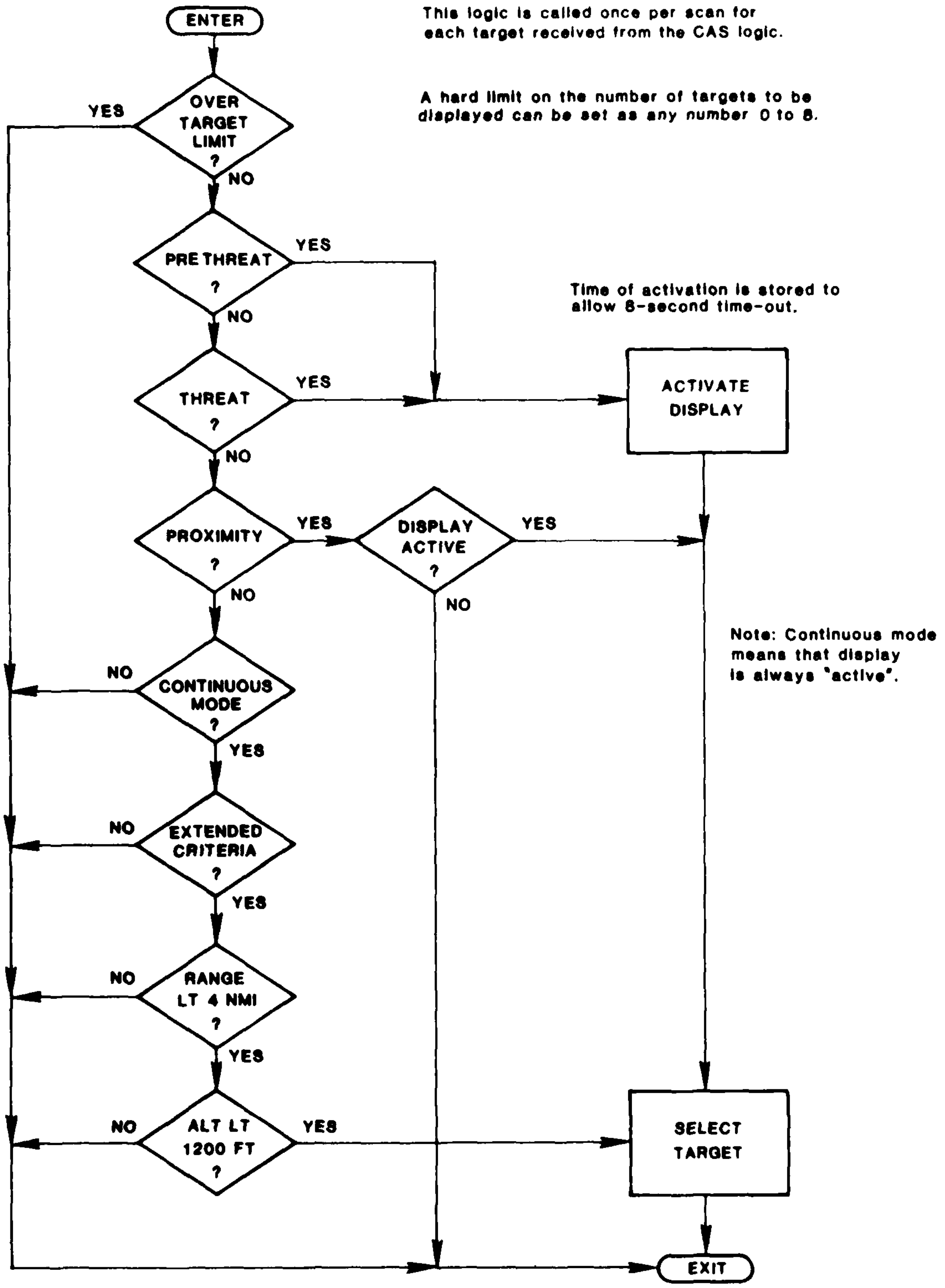

ENTER
This logic is called once per scan for each target received from the CAS logic.
OVER TARGET LIMIT ?
YES
NO
A hard limit on the number of targets to be displayed can be set as any number 0 to 8.
PRE THREAT ?
YES
NO
Time of activation is stored to allow 8-second time-out.
THREAT ?
YES
NO
ACTIVATE DISPLAY
PROXIMITY ?
YES
NO
DISPLAY ACTIVE ?
YES
NO
Note: Continuous mode means that display is always "active".
CONTINUOUS MODE ?
NO
YES
EXTENDED CRITERIA ?
NO
YES
RANGE LT 4 NMI ?
NO
YES
ALT LT 1200 FT ?
NO
YES
SELECT TARGET
EXIT

Fig. C-6. Target selection logic.

This logic is entered once per scan for each target selected by the target selection logic.

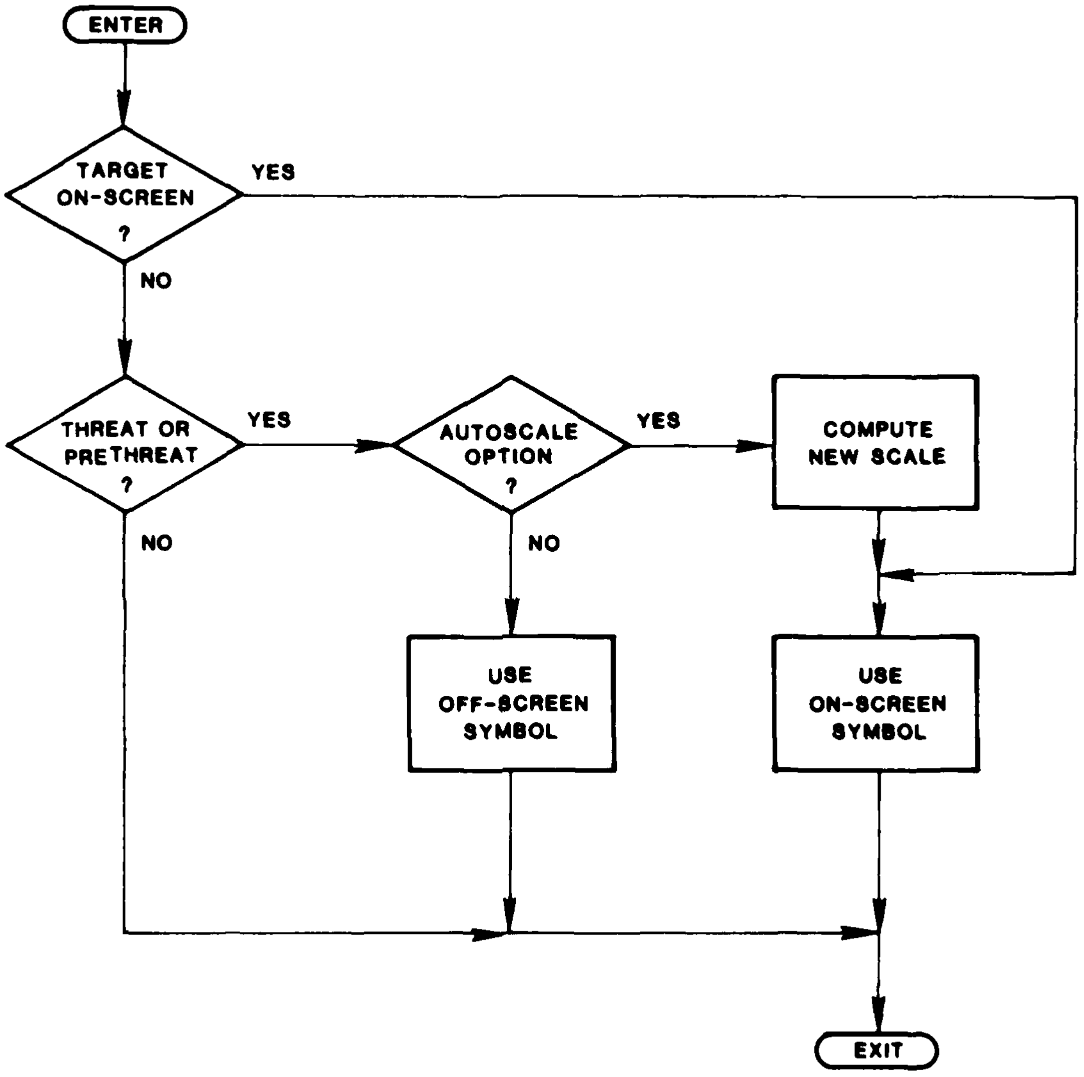


Fig. C-7. On-screen display logic.